ソフトウェア環境のからくりを学ぶ

はじめて読むMASM

蒲地 輝尚 著

アスキー出版局

Photographed by Akio Kambayashi
Designed by Keiko Go

はじめて読む MASM

蒲地輝尚 著

アスキー出版局

● 本書を読む前に ●

本書は,「はじめて読む 8086」の続編であり, 8086CPU, すなわち MS-DOS のアセンブラである MASM と, MASM を使ったプログラム開発全般についての入門書です. 本書の解説は, MS-DOS の動作するすべての機種を対象としています.

前書「はじめて読む 8086」では, コンピュータの最も基礎的な知識である, メモリ, アドレス, 2 進数, 16 進数, ニーモニック, マシン語と CPU などについて, 8086CPU および互換 CPU を対象にやさしく解説しています. これらの基礎知識をまだ学んでいない方は, まずコンピュータ全般の基礎学習から始めてください.

本書は,「はじめて読む 8086」の続編ではありますが, 基礎学習は任意の参考書を選んでもかまいません. また, 8086CPU の特徴であるセグメント方式を理解している必要はありません. セグメント方式に関しては, 本書でくわしく解説します.

本書の実行例などで利用している MASM のバージョンは 3.00 ですが, バージョン 2.40 以降の MASM であれば, 本書の解説はすべて当てはまります. なお, MS-DOS のバージョンは 2.11 以降を対象としています(2.11 より古いバージョンはほとんど使われていない). メーカーによっては MASM が MS-DOS とは別売になっている場合もあり, この場合は別途購入する必要があります.

実習のためのサンプルとして取り上げたシステムは NEC の PC-9801 シリーズ用の MS-DOS ですが, 6.2 章の実習を除いて, どの機種でも本書のプログラムをそのまま実行することができます. 6.2 章のプログラムについては, 富士通の FMR-60/70 用の例も紹介しています.

はじめに

私がマシン語プログラムの作成を初めて体験したのは，ある8ビットのパーソナルコンピュータです．エディタ，アセンブラ，デバッガといったプログラムが1つになったエディタアセンブラというツールを利用しました．それぞれの機能はとてもシンプルで，わかりやすいものでした．ちょっとしたプログラムを入力してはアセンブルして実行し，結果を確認するという実験を繰り返してコンピュータに対する理解を深めたものです．

パーソナルコンピュータの能力が向上するにつれ，オールインワンタイプのツールはそれぞれの機能を大幅に強化した専用ツールに移行しました．パーソナルコンピュータ自身を使ってアプリケーションを開発する，プロの要求を満足する環境に近づいていったといってよいでしょう．反面，私たちのようなアマチュアが趣味としてプログラムを作成するには，少々面倒な仕組みになったことも確かです．

しかし，プロの道具であってもマシン語プログラムの作成が目的であることに変わりはありません．中心となる概念や機能さえ理解すれば，十分活用できます．さらに贅沢なことに，プロの手になじんだ本格的な付加機能までも利用することができるのです．

前書「はじめて読む8086」ではコンピュータの仕組みや，コンピュータを直接操作できるマシン語についてわかりやすく解説しました．コンピュータの真の姿を理解するにつれ，マシン語でプログラムを組んでみたくなった方も多いことでしょう．また，8ビットマシンでマシン語プログラミングを体験したことがあり，16ビットのMS-DOSでも挑戦してみたいという方もいるでしょう．

MS-DOSではマシン語プログラムを開発する道具(ツール)として，MASMというアセンブラが提供されています．ところが，MASMは高機能であるとともに，難解であるとも思われています．しかし，マシン語命令もアセンブラの文法もCPUによって大きく異なるということはありません．ニーモニックが多少異なるくらいでだいたい似たようなものです．8086CPUの場合も例外ではなく，他のCPUでの経験があれば容易にプログラムを組むことができるでしょう．

8086CPUのマシン語が難しいと思われている理由は，セグメント方式にあります．この点だけは他のCPUにない特徴です．MASMは8086CPUの機能をフルに活かせるだけの能力を持っており，当然セグメントを扱う手段を備えています．8086CPUのセグメント方式は8ビットCPUとの互換性にも大きな利点があるわけですが，8ビットCPU的なメモリモデルでもセグメントに関する設定が必要となるところにとまどいがあるようです．16ビットCPUとしての能力を活かして大容量のメモリを利用しようとすると，当然さらにくわしい知識を必要とします．

本書ではMASMのアセンブラとしての一般的な機能を解説するとともに，セグメント方式の仕組みや扱い方を理解すべき順番に整理してやさしく解説します．順を追って確認していけば，必ずすっきりとした概念を頭のなかに構築することができます．セグメントは言われているほど難しいものではないのです．

マシン語プログラムを開発する道具は，アセンブラだけではありません．アセンブラを中心とする種々のツール群の連携があってこそプログラムを作成できるのです．本書のタイトルは「はじめて読むMASM」ですが，MASMだけではなく，MASMを中心とするプログラミング環境についても解説しています．

現在ソフトウェア開発にはC言語を中心とする環境が広く使われていますが，その基本的な仕組みはアセンブラを中心とするプログラミング環境と同一です．各ツールの本当の役割は，アセンブラによるプログラム開発を体験して初めて理解できるものでしょう．高級言語の利用者にも本書を通じてプログラム開発のエッセンスを吸収してもらえることを期待します．

1988年7月　　蒲地輝尚

CONTENTS

はじめに……3

1 ▶アセンブラをはじめる前に――11

1.1 アセンブラの位置づけ……13

1.2 マシン語プログラムの開発手順……17

マシン語プログラムの開発に必要なツール 18

ソースプログラムの入力 19

アセンブル 20

リンク 22

COM形式への変換 23

プログラムの実行 24

2 ▶アセンブラをとりまく世界――25

2.1 ハードウェア環境……27

マシン語命令の世界 27

ハードウェアの仕組み(1)―ディスプレイ画面の制御 29

ハードウェアの仕組み(2)―その他の周辺機器の制御 30

8086CPUにおけるマシン語命令の世界 31

マシン語命令によるプログラミングの世界 31

2.2 ソフトウェア環境……35

ROM BIOS 35

ハードウェアの違いを吸収するMS-DOS 37

高機能な入出力を提供するMS-DOS 38

システムコール 40

入出力方法の選択 44

2.3 プログラミング環境としてのアセンブラ……52

アセンブラの世界でのプログラミング 52

アセンブラの守備範囲 53

COLUMN ハードウェア情報の入手方法 34

エスケープシーケンスによる画面制御 48

MS-Windows, OS/2のマルチウィンドウ環境 50

IRET, CLI, STI命令―割り込み処理に関するマシン語命令― 57

Illustration:南保なおみ

CONTENTS

3 ▶アセンブラ・プログラミングの基礎 —— 59

3.1 アセンブラとマシン語 …… 61
マシン語命令と擬似命令 61
擬似命令の役割 64

3.2 数値表現とラベル …… 65
数値表現 66
ラベル 66

3.3 ORG擬似命令とEND擬似命令 …… 69
ORG擬似命令 69
END擬似命令 71

3.4 データ定義擬似命令 …… 73
DB擬似命令 73
DW擬似命令 76
その他のデータ定義命令 77

3.5 データラベル …… 78
データラベルの定義 78
データラベルの参照 79
配列の参照 80
レジスタを使った配列の参照 81

3.6 OFFSET演算子とPTR演算子 …… 82
OFFSET演算子 82
PTR演算子 83
データラベルの前方参照 85
データ型の強制的な変更 86

3.7 SEGMENT擬似命令とASSUME擬似命令 …… 87
SEGMENT～ENDS擬似命令 87
ASSUME擬似命令 88

3.8 読みやすいプログラムを書くために …… 89
フィールド 89
コメント 90

3.9 COMモデルのプログラム実習……93
アセンブルの操作 93
アセンブル(MASMコマンド) 93
リンク(LINKコマンド) 97
ファイル変換(EXE2BINコマンド) 98
デバッガ(SYMDEBコマンド) 98
ラベルの効果 100
ORG擬似命令の効果 102
DB擬似命令の効果 102
COLUMN MASMのバージョンによるエラーメッセージの違い 96
NOP命令ー何もしない命令?ー 103

4 セグメントの本格的活用――105

4.1 セグメントの概念……107
セグメントに関する知識の必要性 107
利用可能なメモリ容量 110
セグメント 112
セグメントの大きさ 113
セグメントアドレスとオフセットアドレス 114
4.2 セグメント方式の仕組み……116
プログラムの実行の仕組みとセグメント 116
セグメントレジスタの役割 118
セグメントの選択方法 120
物理アドレスとセグメントアドレス 123
4.3 SEGMENT擬似命令……127
セグメントの定義 127
セグメントごとに名前を付ける 129
スタックセグメントの定義 130
4.4 ASSUME擬似命令……132
ASSUME擬似命令の役割 132
データセグメントの選択 133
コードセグメントとASSUME擬似命令 136

CONTENTS

4.5 GROUP擬似命令 ……138
セグメントのグループ化 138
GROUP擬似命令の書式 139
セグメントグループとASSUME擬似命令 139
4.6 セグメントを使いこなす ……142
セグメントオーバーライドプリフィックス 142
セグメントオーバーライドプリフィックスの自動挿入 144
暗黙のセグメント指定(ローカル変数) 147
GROUP擬似命令とOFFSET演算子 152
4.7 EXEモデルのプログラム実習 ……153
標準入出力をバッファリングするプログラム 153
アセンブルの手順 158
EXEモデルのプログラム実行開始時のセグメントレジスタ 160
SEGMENT擬似命令，END擬似命令の効果 161
ASSUME擬似命令，GROUP擬似命令の効果 164
セグメントオーバーライドプリフィックスの確認 166
セグメントのリロケート 167
4.8 EXEファイルの構造と仕組み ……170
EXEファイルの構造 170
COMモデルとEXEモデルの違い 172
PSP 173
COLUMN LODS, STOS, SCAS, MOVS, CMPS命令ーストリング命令ー 150

5 ▶ マクロアセンブラとモジュール別プログラミング ――175

5.1 プログラミングを効率化する擬似命令 ……177
EQU擬似命令 177
INCLUDE擬似命令 180
PROC～ENDP擬似命令 181
PTR演算子(2) 190
IF擬似命令 190

5.2 マクロアセンブラとは ……195
ハンドアセンブル 195
1ラインアセンブラ 196
2パスアセンブラ 197
マクロアセンブラ 197
5.3 マクロ命令とEQU擬似命令 ……199
マクロ命令とは 199
EQU擬似命令(2) 200
5.4 MACRO擬似命令 ……204
マクロ定義とマクロ呼び出し 204
マクロ展開の仕組み 205
マクロパラメータ 207
パラメータによる条件マクロ 209
LOCAL擬似命令 210
マクロ命令とプロシージャの違い 211
5.5 分割アセンブルの概念 ……216
分割アセンブルとは 216
分割アセンブルの利点 218
5.6 PUBLIC擬似命令とEXTRN擬似命令 ……222
PUBLIC擬似命令 222
EXTRN擬似命令 223
セグメントの結合(コンバインタイプ) 226
5.7 分割アセンブルの手順とライブラリ機能 ……228
分割アセンブルの手順 228
分割アセンブルの仕組み 235
リンカのライブラリ検索機能 237
ライブラリアンの働き 239
COLUMN アセンブラを構造化するマクロ 214

CONTENTS

6 ▶ アセンブラ実用テクニック —— 241

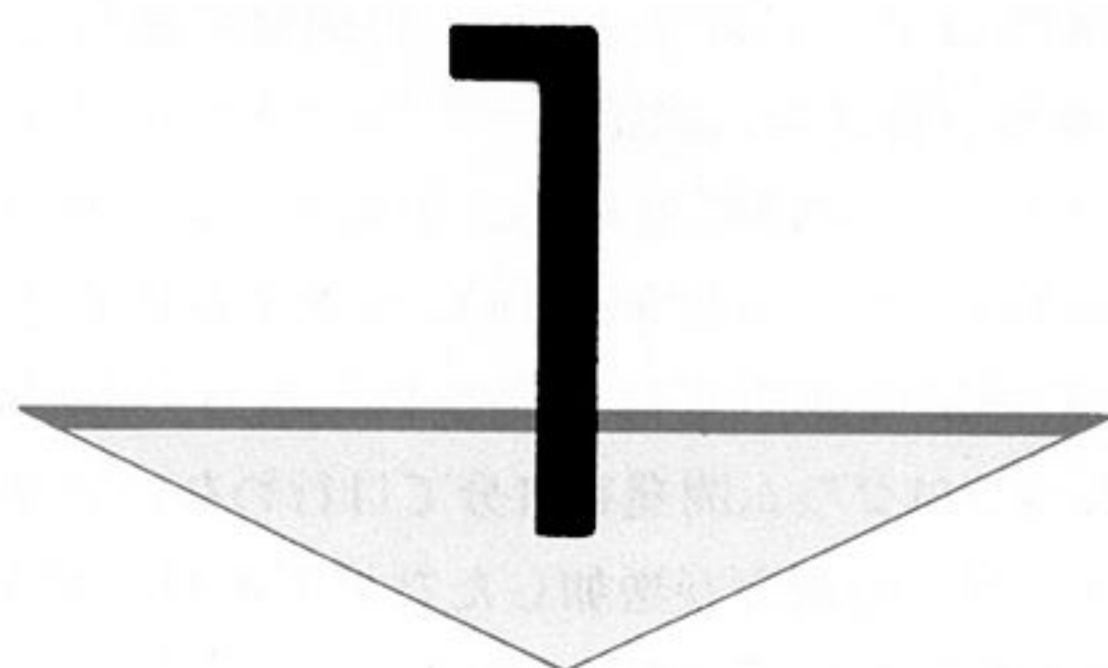
6.1 C言語とのリンク ……… 243
アセンブルとコンパイル 243
関数=PROC 244
C言語からMASMへの引数の受け渡し 245
MASMからC言語への値の返し方 248
スモールモデルとラージモデル 249
MASMとC言語をリンクするための約束事 253
コンパイラによるアセンブラソースの出力 254
ヘッダファイルの利用 256
サンプルプログラム―オセロゲームの思考ルーチン 257

6.2 ハードウェア割り込み ……… 272
割り込みプログラムの登録 273
システムの初期化 273
割り込み処理ルーチン 275
サンプルプログラム―スクリーンセーバー 277

6.3 デバイスドライバ ……… 290
デバイスドライバの概念 291
デバイスドライバの利用方法 294
サンプルプログラム―ローマ字カナ変換デバイスドライバ 296

COLUMN LDS, LES命令―ポインタをロードするマシン語命令― 252
C言語とアセンブラの統合 270

APPENDIX ▶ —— 301
80286CPUの機能とMS-OS/2 302
MASMコマンドの使い方 310
LINKコマンドの使い方 312
LIBコマンドの使い方 315
MASM擬似命令一覧 318
MS-DOS主要ファンクション一覧 322

索引 ……… 331
参考文献 ……… 335

1 アセンブラをはじめる前に

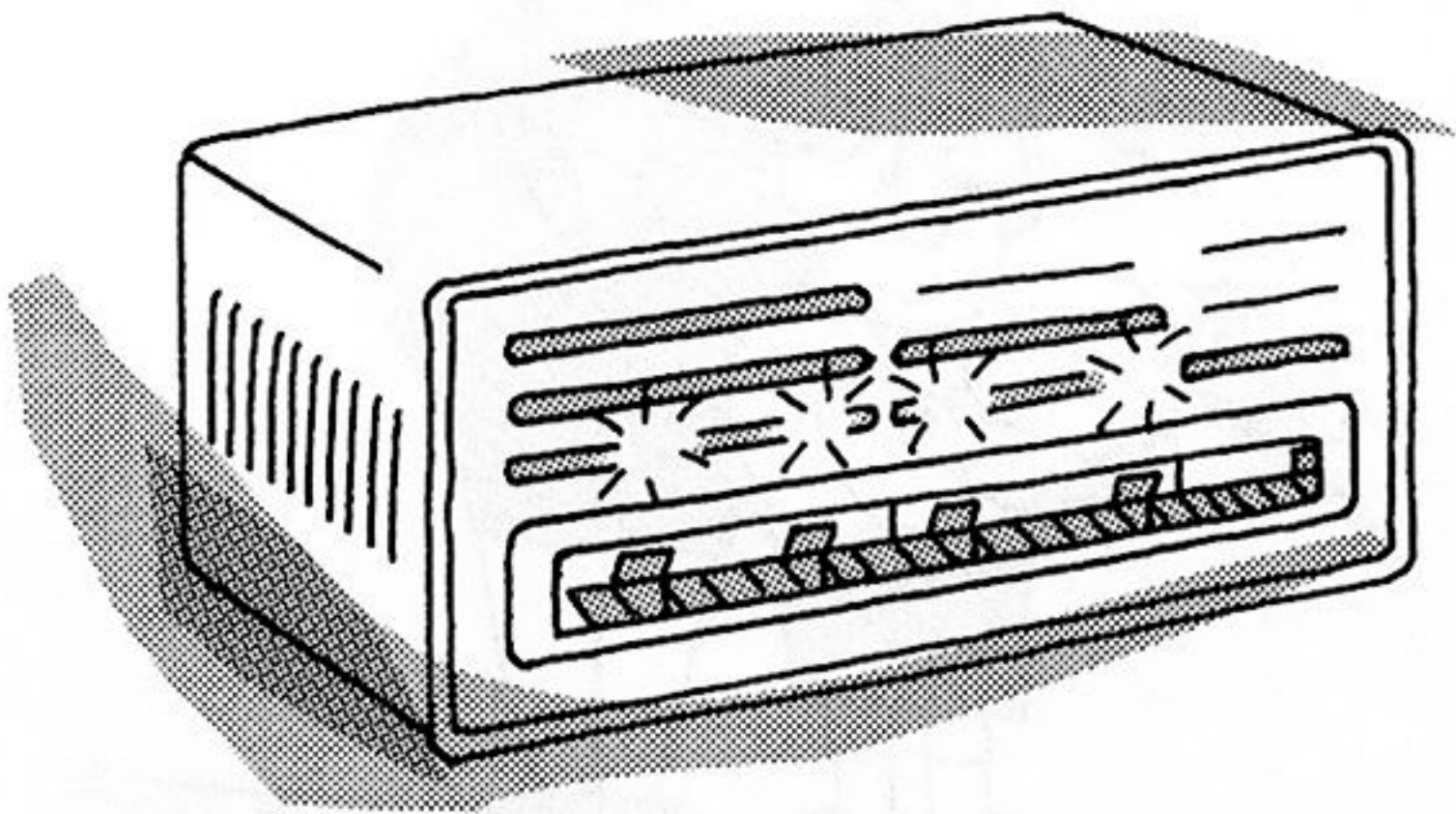
MS-DOS はマシン語プログラムの開発に適した環境です.なぜならプログラム開発ツール(ソフトウェアを作成する道具)として,MASM を始めとするコマンド群が提供されているからです.当初 MS-DOS システムに必ず付属していた MASM も最近では別売されるようになりましたが,これはプログラム開発を自分では行わないアプリケーションユーザーの割合が増加したためであり,MASM の役割が失われたからではありません.

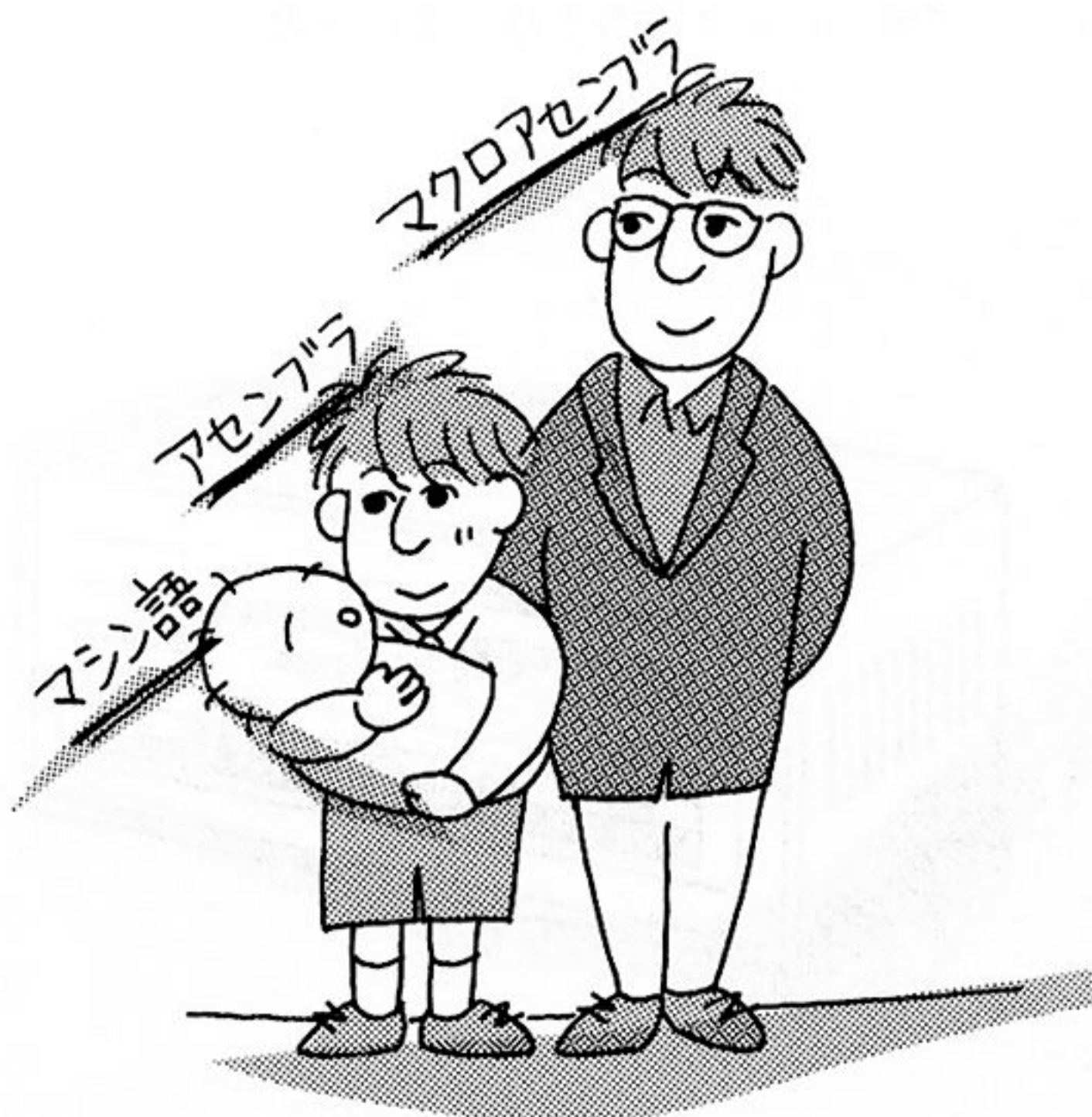
本章では,プログラム開発におけるアセンブラの位置づけを歴史的な流れから概観していきます.そのあとで,アセンブラによるプログラム開発を実際に体験してみることにします.

1.1
アセンブラの位置づけ

最も初期のコンピュータにおけるプログラムは，電気的な回路，つまり固定的な配線として作成されました．したがってプログラムを変更したり改良したりするためには，配線のつなぎ方を変えなければなりませんでした．

やがてプログラムを書換え可能なメモリに格納し，配線を変更しなくてもいろいろなプログラムを動かせるようになりましたが，それでもまだたいへんでした．たとえば，プログラムを次のような方法で入力していたのです．

一列に並んだスイッチのON／OFFをビットパターンとみなして，メモリのアドレスをセットします．同様にメモリに書き込むデータもデータ用のスイッチ列にセットします．この状態で書き込みボタンを押すことにより，指定したアドレスのメモリにデータが書き込まれます．こうしてマシン語プログラムを1ステップずつメモリに書き込んでいったのです．

ここまでの解説でわかるように，初期のプログラムはマシン語そのもので，つまりビットパターンや数値で表現されていたのです．しかしそれではプログラムを理解したり表現したりするのにあまりにも不便なので，1つ1つのマシン語を記号で表すようになりました．これがアセンブリ言語のニーモニックです．ニーモニックを使ってプログラムをアセンブリ言語で記述することにより，プログラムの開発効率は向上しました．そのかわり，ニーモニックをマシン語コードに変換するという操作が必要になります．この操作がアセンブルです．

アセンブルは非常に単純な作業であり，それこそコンピュータによって自動化されるべき仕事です．そのために開発されたのがアセンブラであり，やがてアセンブラにはプログラム作成を容易にするためのさまざまな機能が追加され，プログラムの開発にはなくてはならないものとなりました．その1つがマクロ機能です．マクロ機能についてはあとの章でくわしく解説しますが，プログラムの書きやすさを飛躍的に向上させるものです．

もう1つの大きな流れとして，高級言語が開発されたことに触れる必要があるでしょう．マシン語命令では加算や減算のような単純な演算しかできず，複雑な計算を行うためにはいくつもの命令を組み合わせなければなりません．高級言語は，数学的な式の形で計算方法を書いておくと，自動的にマシン語命令の組合せに置き換えてくれるプログラムとして開発されました．初期のプログラミング言語の代表であるFORTRAN（FORmula TRANslation：数式変換プログラム）の名前の由来がそのことをよく表しています．

また高級言語では条件分岐やループなどのプログラムの基本構造を一定の書式で書き表すことで，アリゴリズムをそのまま表現できるようになりました．高級言語を利用すれば，プログラムがわかりやすくなりデバッグなども容易になります．さらにCPUが進歩することによって，マシン語命令の構成が変わってもプログラムを変更しなくてもよいという大きなメリットがあります．

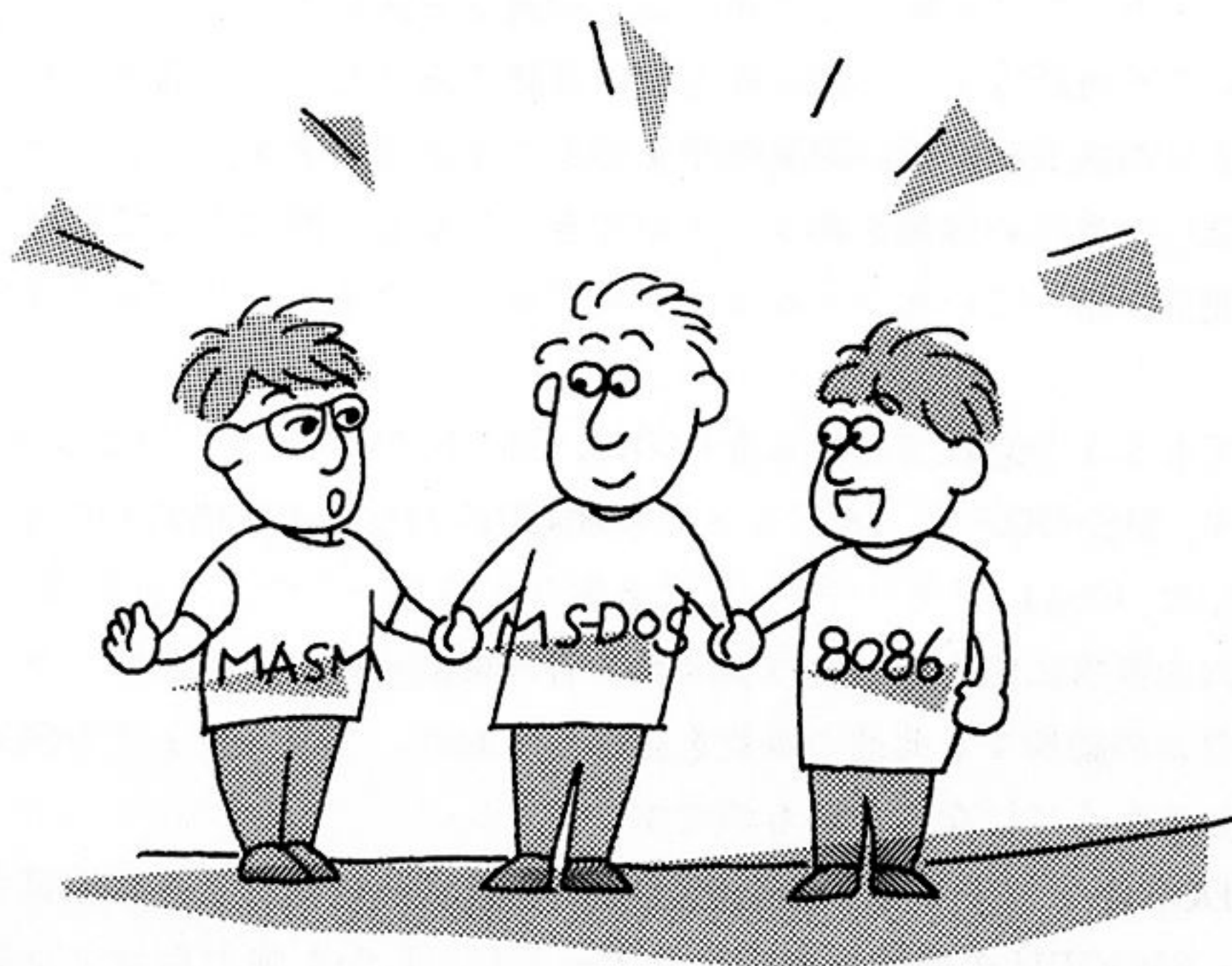
しかし，高級言語によってアセンブラの役割が失われてしまうわけではありません．一般的なプログラムならば，高級言語を使って開発するべきでしょうが，CPUやハードウェアの機能を十分に利用するためにはCPUの動作に直結したアセンブラを使うことがやはり最適な方法です．

アセンブリ言語はマシン語に最も近い言語でありながら，前述したマクロ機能により高級言語に近い開発効率を得ることができます．マシン語という機械の側に一番近い言葉を扱うことができ，しかも人間にとって扱いやすい機能を豊富に持っているツールはマクロアセンブラをおいてほかにはないでしょう．

そしてもう1つ忘れてはならないのは，OS（オペレーティングシステム）の登場です．MS-DOSは，16ビットの8086CPUのための本格的OSとして生まれました．OSは，メモリやディスク装置などのハードウェアを管理し，各プログラムの要求に応えるというシステム管理機能を持っています．さらに，プログラムの動作する共通の基盤を提供しており，ソフトウェアを開発する道具(ツール)としてなくてはならないものです．

MS-DOSをソフトウェア開発環境という観点から見たときに注目すべきことは，8086CPUの機能を十分にサポートし，しかも強力なマクロ機能を持ったアセンブラ「MASM」が提供されていることです．コンピュータの仕

組みを深く理解するためには，CPU がマシン語によって動作する仕組みを理解することが必要であることを前書「はじめて読む 8086」で解説しましたが，私たちはマシン語によるプログラムの作成に MASM という高度なツールを利用することができるのです．

本書では MASM の役割を解説しながら，同時に 8086CPU の機能とマシン語プログラム開発におけるさまざまな概念を解説していきます．

1.2

マシン語プログラムの開発手順

MASMでの，すなわちアセンブリ言語のプログラムの書き方を解説する前に，まずMASMを使ったプログラム開発の手順を解説しましょう．MASMによるマシン語プログラムの開発は，図1-1に示すような手順で行われます．

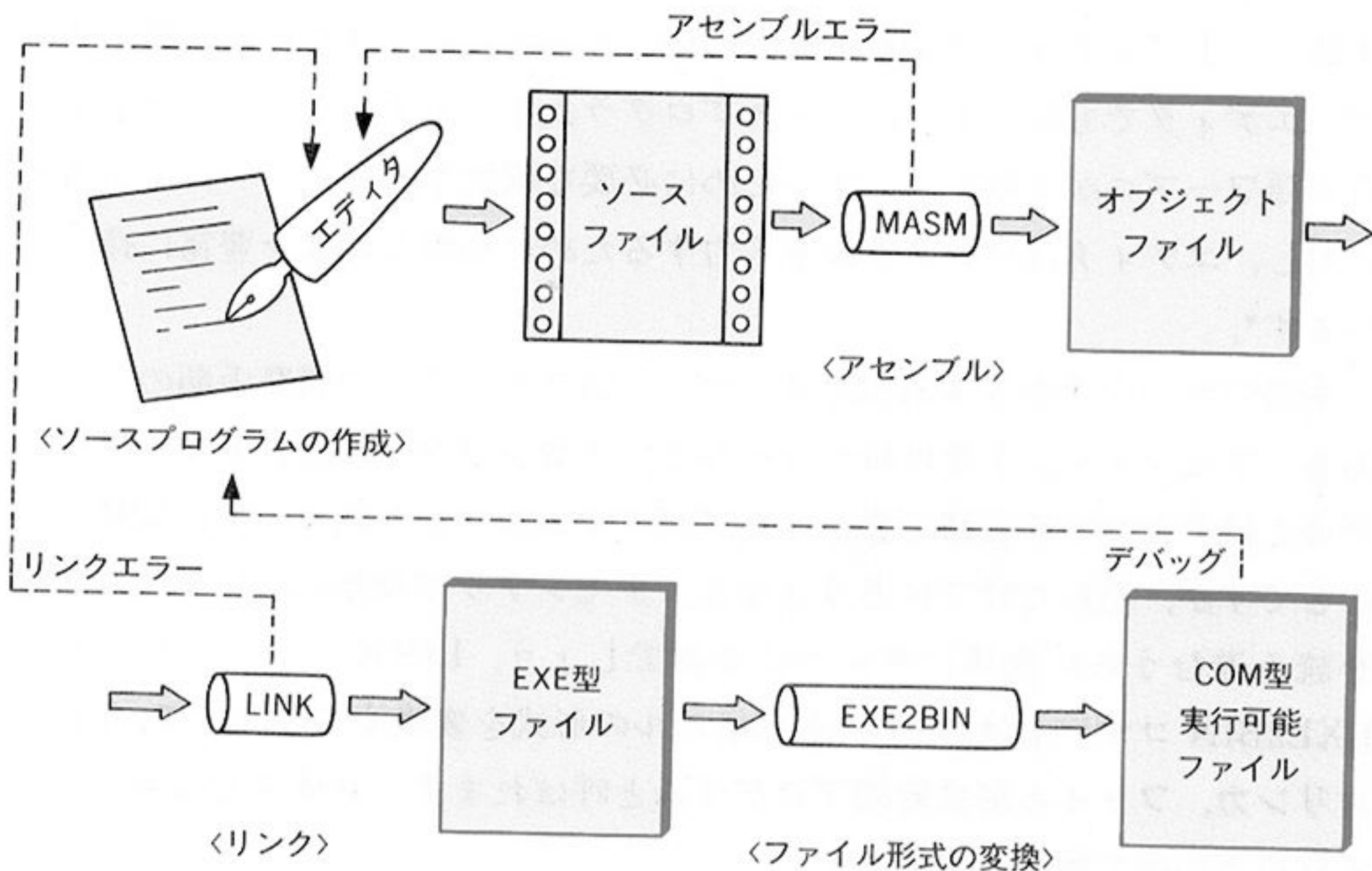



図1-1　MASMによるマシン語プログラムの開発手順

プログラムの開発は図に示すように4つの処理からなり，それぞれ異なった役割を持っています．この図からは，いくつもの手順に分かれている理由がわからないかもしれません．中間的なファイルがいくつも生成されてむだなように思えるでしょうが，それにはちゃんとした理由があるのです．くわしくはあとの章で解説しますので，ここでは一連の4つの処理が必要であることを理解してください．

マシン語プログラムの開発に必要なツール

マシン語プログラムの開発には，少なくとも以下に示すツールが必要です．これらのツールは先に解説した4つの処理にそれぞれ対応しています．

任意のエディタ	…市販のスクリーンエディタ*
MASM.EXE	…アセンブラ
LINK.EXE	…リンカ
EXE2BIN.COM	…ファイル形式変換プログラム

エディタはソースプログラムを入力するツールです．読者のみなさんは日本語ワードプロセッサで文章を入力したことがあるのではないかと思いますが，エディタでも同じようにソースプログラムを入力することができます．日本語ワープロが文章を入力するために必要な機能を持ったツールであるのに対し，エディタはプログラムを入力するために必要な機能を豊富に持っています*．

本書のテーマであるMASMは，マシン語プログラムの開発手順の1つである「アセンブル」を受け持つツールで，**アセンブラ**と呼ばれます．アセンブルとはアセンブリ言語で書かれたプログラムをマシン語コードに変換することですが，それだけではありません．アセンブラの役割については，本書を読み進むうちに次第に明らかになるでしょう．LINKコマンドおよび，EXE2BINコマンドは，いずれもファイルの形式を変換するツールで，それぞれ**リンカ**，**ファイル形式変換プログラム**と呼ばれます．具体的な役割については以下の項で解説します．

MASMのオリジナルディスクには，**図1-2**に示すように必要なツールが含まれています．ただし，エディタに関しては，豊富な機能を持った市販のものを購入して利用することをお勧めします．

なお，マシン語プログラム開発の実習は，必ずオリジナルディスクのバックアップをとり，作業用ディスクを作成してから行ってください．

---

*日本語ワープロでもMS-DOSのテキストファイルを出力できるものならば，ソースプログラムを作成するためのエディタとして利用することができる．

MASM Ver3.0 (NEC製)

```
A>DIR ↵

ドライブ A: のディスクのボリュームラベルはありません.
ディレクトリは A:¥
                                         アセンブラ
MASM     EXE    77362  86-10-13   0:00
CREF     EXE    10544  86-10-13   0:00
LINK     EXE    41114  86-10-13   0:00
LIB      EXE    24138  86-10-13   0:00
MAKE     EXE    18675  86-10-13   0:00
MAPSYM   EXE    51904  86-10-13   0:00
SYMDEB   EXE    36538  86-10-13   0:00
        7 個のファイルがあります.               リンカ
    986112 バイトが使用可能です.

A>
```

□は, 本書で使用するコマンド
「EXE2BIN.COM」は, MS-DOSのシステムディスクに付属しているので, それを使用する

MASM Ver4.0(マイクロソフト製)

```
A>DIR ↵

ドライブ C: のディスクのボリュームラベルはありません.
ディレクトリは C:¥

MASM     EXE    87124  87-01-26    4:00
LINK     EXE    43988  85-10-16    4:00
SYMDEB   EXE    37021  85-10-16    4:00
MAPSYM   EXE    18026  85-10-16    4:00
CREF     EXE    15028  85-10-16    4:00
LIB      EXE    28716  85-10-16    4:00
MAKE     EXE    24300  85-10-16    4:00
EXEPACK  EXE    10848  85-10-16    4:00
EXEMOD   EXE    11034  85-10-16    4:00
COUNT    ASM     5965  85-10-16    4:00
README   DOC     7630  85-10-16    4:00
EV       ORG    43010  87-02-26   15:25
INSTJ    EXE    20580  87-02-26   15:25
INSTA    EXE    20492  87-02-26   15:25
EV       EXE    43010  87-02-26   15:25
        15 個のファイルがあります.          エディタ
    824320 バイトが使用可能です.

A>
```

図1-2 MASMのオリジナルディスクに含まれるファイル

ソースプログラムの入力

まず最初に，ソースプログラムを作成します．ソースプログラムの作成に必要なエディタの使い方は，本書では解説しません．自分の使っているエディタのマニュアルや解説書を参考にしてください．プログラム例を**リスト1-1**に示します．ファイル名を「OTENKI.ASM」として，エディタを使って入力してください．

このプログラムは，天気を予測するプログラムです．あなたの持つ超能力によって制御される微妙なタイミングを測り，その結果からお天気を推測します*．非常に短いプログラムなので，MASMによるマシン語プログラム開発の手順を体験するためにもぜひ入力してみてください．

---

*もし，このプログラムの予報がはずれたとすると，あなたに超能力が足りないということです．

リスト 1-1　天気予報プログラム OTENKI.ASM

```
        ASSUME  CS:CODE,DS:CODE
CODE    SEGMENT

        ORG     100H
START:  MOV     BX,0
NOINPUT:MOV     AH,06H
        MOV     DL,0FFH
        INT     21H
        JNZ     PRINT
        INC     BX
        CMP     BX,5
        JGE     START
        JMP     NOINPUT
PRINT:  SHL     BX,1
        MOV     DX,TABLE[BX]
        MOV     AH,09H
        INT     21H
        MOV     AH,4CH
        MOV     AL,00H
        INT     21H

HARE    DB      'HARE',0DH,0AH,'$'
KUMORI  DB      'KUMORI',0DH,0AH,'$'
AME     DB      'AME',0DH,0AH,'$'
ANOTI   DB      'AME NOTI HARE',0DH,0AH,'$'
KNOTI   DB      'KUMORI NOTI AME',0DH,0AH,'$'

TABLE   DW      OFFSET HARE,OFFSET KUMORI,OFFSET AME
        DW      OFFSET ANOTI,OFFSET KNOTI

CODE    ENDS
        END     START
```

アセンブル

［コマンド形式］　MASM　ファイル名；

マシン語プログラム開発の第2のステップ，アセンブルの方法を解説します．リスト1-1に示したソースプログラムはアセンブリ言語で記述してありますが，これをマシン語コードに変換するのが図1-3の操作です．

ソースファイル「OTENKI.ASM」をアセンブルすると，オブジェクトファイル「OTENKI.OBJ」が生成されます．ファイル拡張子「OBJ」は，OBJect（オブジェクト）の略です．

```
A>DIR OTENKI.ASM

 ドライブ A: のディスクのボリュームラベルは WORK
 ディレクトリは A:¥SRC

OTENKI    ASM        538  88-05-15  17:27   ……………ソースファイルOTENKI.ASM
          1 個のファイルがあります.
   958464 バイトが使用可能です.
                                        MASMを実行して，OTENKI.ASMをアセンブルする．
A>MASM OTENKI;  ……………………………………コマンドラインの最後に「;」(セミコロン)を付けること
Microsoft MACRO Assembler  Version 3.00
(C)Copyright Microsoft Corp 1981, 1983, 1984


19374 Bytes free

Warning Severe
Errors  Errors    エラーがないというメッセージ
0       0

A>DIR OTENKI.*

 ドライブ A: のディスクのボリュームラベルは WORK
 ディレクトリは A:¥SRC

OTENKI    ASM        538  88-05-15  17:27
OTENKI    OBJ        190  88-07-17   0:21   ……………オブジェクトファイル
          2 個のファイルがあります.                 OTENKI.OBJが生成された
   957440 バイトが使用可能です.

A>
```

図1-3　アセンブルの操作

ソースプログラムにミスがあると，アセンブルエラーが発生します．例題のプログラムは正しくアセンブルできるはずですが，入力の際にタイプミスなどがあるかもしれません．アセンブルエラーの対処法については3章で解説しますが，エラーが発生すると次の手順へ進めませんので，ソースファイルをリスト1-1と照らし合わせて入力ミスを修正してください．

リンク

［コマンド形式］　LINK　ファイル名；

オブジェクトファイルはCPUが直接実行できるマシン語コードを含んでいますが，そのまま実行できる形式にはなっていません．これをMS-DOSのコマンドとして実行できる形式に変換するのがLINKコマンドです．図1-4のようにLINKコマンドを実行することにより，OBJ型ファイルからEXE型ファイルが生成されます．

```
A>LINK OTENKI;↵ ……………LINKを実行する．コマンドラインの最後に「;」(セミコロン)を付けること

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

Warning: no stack segment ……………この警告は，ここでは気にしなくてよい

A>DIR OTENKI.*↵

 ドライブ A: のディスクのボリュームラベルは WORK
 ディレクトリは A:¥SRC

OTENKI   ASM     538  88-05-15  17:27
OTENKI   OBJ     190  88-07-17   0:21
OTENKI   EXE     869  88-07-17   0:23 …………EXE形式のファイルOTENKI.EXEが生成された
         3 個のファイルがあります.                  (このファイルは実行できない)
    956416 バイトが使用可能です.

A>
```

図1-4　リンクの操作

LINKは「つなぐ」という意味であり，LINKコマンドの本来の役割は複数のオブジェクトファイルをつなげて1つの実行可能ファイルを生成することです．この仕組みについては5章で解説しますが，ここではオブジェクトファイルを実行可能な形式のファイルに変換するために必要な操作であることを理解してください．

なお，ここで重要な注意があります．MS-DOSの実行可能ファイルの形式には，ファイル拡張子で区別されるCOM形式とEXE形式があり，プログラ

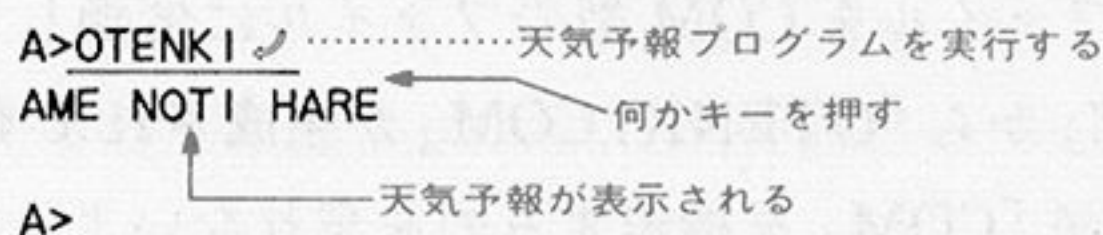
ムの構造も異なります*．OTENKIプログラムはCOM形式として作成しているため，EXE形式では実行できません．もしそのまま実行すると暴走する危険がありますので，もう少し待ってください．

COM形式への変換

［コマンド形式］　EXE2BIN　ファイル名　ファイル名.COM

EXE形式として作成したプログラムは，リンクしたあと，そのまま実行することができます．しかしCOM形式のプログラムは，EXE形式からCOM形式へファイルを変換しなければなりません．図1-5に示すように，EXE2BINコマンドでEXE型のファイルをCOM型のファイルに変換します．これにより，「OTENKI.EXE」から「OTENKI.COM」が生成されます．変換後のファイル名として，拡張子「COM」を指定するのを忘れないようにしてください．

```
A>EXE2BIN OTENKI OTENKI.COM ⏎ ……………EXE2BINを実行して，ファイル形式を変換する

A>DIR OTENKI.* ⏎

 ドライブ A: のディスクのボリュームラベルは WORK
 ディレクトリは A:¥SRC

OTENKI   ASM     538  88-05-15  17:27
OTENKI   OBJ     190  88-07-17   0:21
OTENKI   EXE     869  88-07-17   0:23
OTENKI   COM     101  88-07-17   0:25 ……………COM形式の実行可能ファイルOTENKI.COM
        4 個のファイルがあります.                 が生成された
   955392 バイトが使用可能です.

A>
```

図1-5　ファイル形式の変換

*ファイル拡張子EXEは「EXEcutable」の略で，実行可能という意味．ファイル拡張子COMは「COMmand」の略で，実行可能なコマンドという意味．拡張子の名前は歴史的な経緯から決められたもので，プログラムの構造とは関係がない．

プログラムの実行

［コマンド形式］　ファイル名

できあがった「OTENKI.COM」は，MS-DOSの他のコマンドと同じようにファイル名を入力するだけで実行できます．その様子を図1-6に示します．OTENKIコマンドを実行すると，キー入力待ちの状態になります．そこで精神を集中し雑念を捨て，気合いを込めてキーボードをたたくと，お告げが表示されます．

図1-6　プログラムの実行

このプログラムは「アセンブラでなければ書けない」というものではありませんが，マシン語プログラムの開発を体験するためには手頃なものといえます．ソースプログラムの書き方については3章でくわしく解説しますので，メッセージを追加したり変更したりして自分なりのプログラムに改良してみることをお勧めします．おみくじプログラムや相性占いプログラムといったバリエーションが考えられるでしょう．

本書では，このプログラムをもとにMASMの機能や役割を解説していきます．さらに，新たなプログラムの作成や改良をしていきながら，8086CPUの機能やそれをサポートするMASMの機能を取り上げていきます．なるべく短いプログラムで興味深い例題を選びましたので，MASMによるプログラム開発をよく理解するためにも自分で入力して試してみてください．

アセンブラをとりまく世界

アセンブラそのものの解説を始める前に，アセンブラでプログラムを作成するために必要な知識をまとめておきましょう．

アセンブラでプログラムを組むには，各種の命令を理解するだけでなく，プログラムがどのような世界で動作するのかを把握しておかなければなりません．アセンブラをとりまく世界は，通常私たちが接しているコンピュータの〝みかけの世界〃ではなく，〝裸のコンピュータの世界〃です．そのために，コンピュータの仕組みや動作原理をはじめとするマシン語に関わる知識は欠かせません．さらに，マシン語プログラムから利用できるハードウェア，ソフトウェア両面にわたる知識も必要です．

とはいっても，コンピュータのすみずみまで完全に理解する必要はありません．おおまかに概念として捉えておけば通常のプログラミングには十分です．本章では，アセンブラでプログラムを作成するための最低限の知識を解説していくことにしましょう．

2.1 ハードウェア環境

アセンブラでプログラムを作成する場合，マシン語命令がプログラムの最小構成要素となります．1つ1つのマシン語命令は，実に単純な機能しか持ちません．しかも，通常のプログラムで利用される命令は，ほんの数十種類といってもいいでしょう．コンピュータのハードウェアはいかにも複雑で種類も多岐にわたるように思えますが，どうしてそれだけの命令で制御できるのでしょうか．本節では，その仕組みを簡単に解説します．

マシン語命令の世界

コンピュータには，ディスプレイ画面やキーボード，ディスク装置など多くの周辺機器が接続されています．しかし，マシン語命令にはそれらを制御するための専用命令などはありません．それどころか，マシン語命令によって操作できるハードウェアは，たったの3種類しかありません．それは，**レジスタ**，**メモリ**，そして**I/O ポート**です(次ページの図 2-1 を参照)．

メモリはご存じのとおり記憶装置です．データを蓄えておき，瞬時に取り出すことができます．マシン語プログラムもメモリに蓄えられます．

レジスタはCPU内部にあるメモリの一種ですが，CPUの動作をコントロールするためのさまざまな役割を持っています．CPU内部にあることから，演算などを高速に行うことができます．

I/O ポートは，CPUと周辺機器をつなぐ橋の役割を持っています．マシン語命令により，CPUはI/O ポートを通して周辺機器に制御信号を送ったり，データを受け取ったりすることができます．

この3つがマシン語命令の世界のすべてです．マシン語命令には，この3つのうちのどれかを操作する命令しかありません．メモリ空間は1バイトのメモリがたくさん並んだものですが，どのメモリも特に区別はなく，同じ命

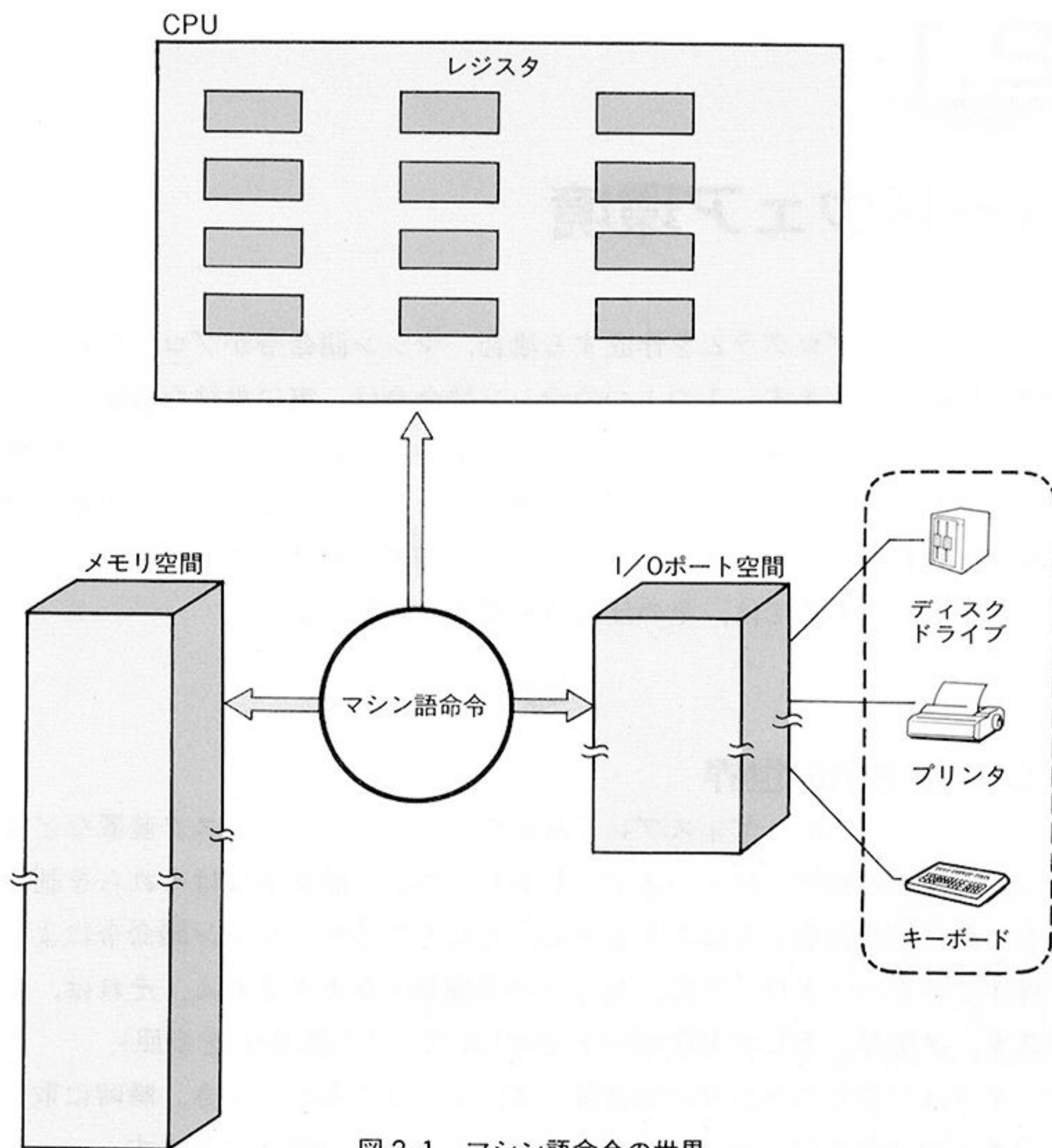



図 2-1　マシン語命令の世界

令で操作します．同様に，I/O 空間は1バイトのI/Oポートがたくさん並んだものですが，どのI/Oポートも同じ命令で操作します．

マシン語命令の世界は，このように実に単純な世界です．周辺機器にはあれだけ多くの種類があるのに，マシン語命令の世界には3つの要素しかありません．たった3つの要素で複雑なハードウェアを制御できるのはどうしてでしょうか．

この疑問に答えるため，わかりやすいように具体的な例を挙げて解説することにします．

ハードウェアの仕組み(1)ーディスプレイ画面の制御

BASICなどの高級言語では,「PRINT "Hello!"」のようなコマンドでディスプレイ画面に文字を表示することができます.ところが,マシン語命令の世界には,ディスプレイ画面に文字を表示するための専用の命令はありません.そのかわり,メモリにデータを書き込むというごく普通の命令によって,ディスプレイ画面に表示される文字を制御します.

多くのマシンでは,メモリ領域のうちのある部分が,VRAM(ブイラム)(Video RAM)と呼ばれる,表示のための特殊な領域になっています.この領域のメモリは,1つ1つが画面上に表示される文字と対応しており,その位置はアドレスによって決まります.たとえば,図2-2のようにアドレスと画面上の位置が対応しています.

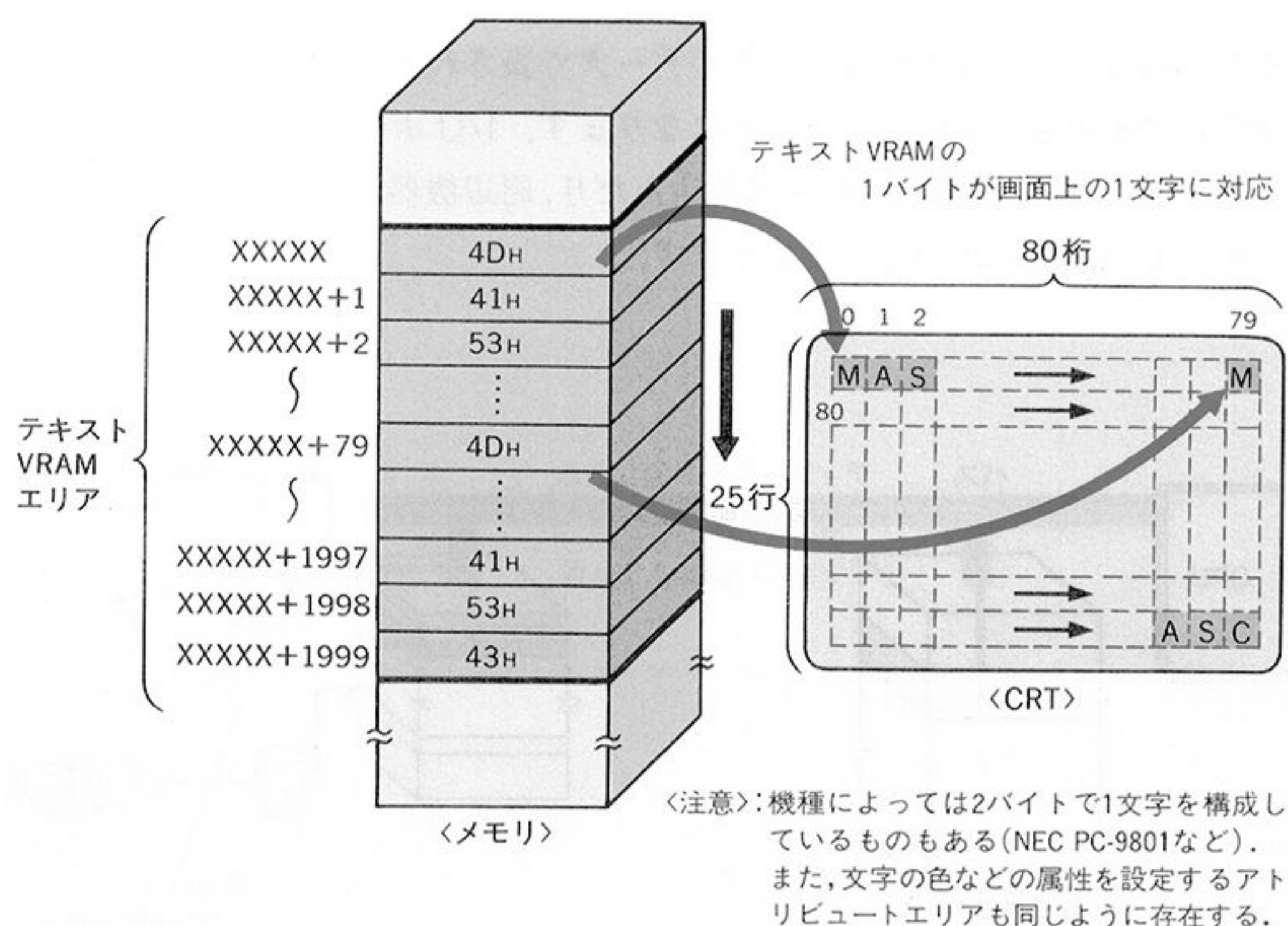



図2-2 VRAMの仕組み

そして，メモリに文字コードを書き込むと，専用のハードウェアによってそのコードに対応する文字が画面に表示されます．メモリの内容を書き換えると，画面に表示される文字も新たに書き込まれた値に応じて変わります．このように，VRAM 領域のメモリに文字コードを書き込むことで画面表示を行うことができます．

ハードウェアの仕組み（2）ーその他の周辺機器の制御

ディスプレイ画面の制御は，周辺機器のなかでも特殊なものであり，他の多くの周辺機器はI/Oポートを通して制御します．図2-3に示すように1つ1つのI/Oポートはそれぞれ，ある特定の周辺機器に接続されています．言い換えれば，1つの周辺機器に1つのI/Oポートのアドレスが割り当てられているといってもよいでしょう．

マシン語命令では，I/Oポートにデータを出力したり，I/Oポートからデータを入力することができます．I/Oポートにデータを出力するということは，そこに接続された周辺機器に，そのデータで表される制御コマンド，あるいはデータそのものを伝達することになります．I/Oポートからデータを入力することは，周辺機器の状態を読み込んだり，周辺機器から送られてきたデータそのものを読み込むことになります．

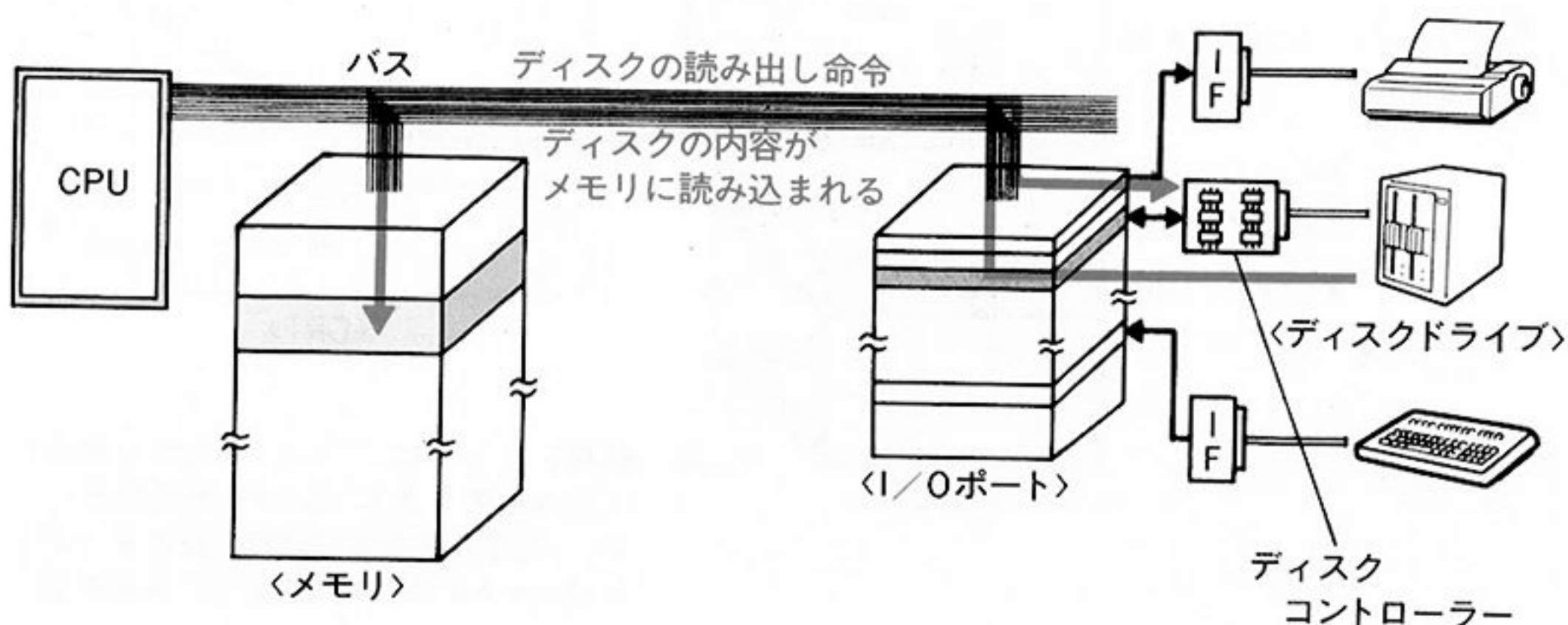



図2-3　I/Oポートと周辺機器

I/Oポートに対して入力および出力することがどういう意味を持つかは，そのI/Oポートに接続された周辺機器によって異なります．たとえば，アドレス××$_H$のI/Oポートからデータを入力することは，キーボードから押されたキーの種類を読み込むことになり，アドレス○○$_H$のI/Oポートにデータを出力するということは，フロッピーディスク装置にデータ転送などの指令を送ることになる，というぐあいです．

図2-3には，フロッピーディスク装置が接続されたI/Oポートに，「ディスクからデータを読み取りメモリに転送する」という指令を表すデータを出力した場合を示しています．

8086CPUにおけるマシン語命令の世界

マシン語命令の世界は，レジスタ，メモリ，I/Oポートの3つから構成されることがわかりました．8086CPUの場合には，次ページの図2-4に示すような世界がマシン語命令の世界になります．図2-4には，8086CPUのレジスタの種類，メモリ空間およびI/O空間の大きさを示しています．

マシン語命令によるプログラミングの世界

アセンブラによるプログラミングは，レジスタ，メモリ，I/Oポートというコンピュータの核になる部分を直接扱うことになるので，BASICなどの高級言語によるプログラミングとは大きく異なります．アセンブラでプログラムを組む場合は，次のことに留意しなければなりません．

第一に，周辺機器のコントロールには，対象とする機器に関する詳細な知識が必要です．たとえば，ある周辺機器は次のような手順で制御しなければなりません．まず，機器の状態をI/Oポートから読み込みます．読み込んだデータの各ビットは，機器が指令を受け付ける状態にあるか，前回の指令をまだ実行中であるかなどの状態を表しています．ビットをチェックすることにより，指令やデータを受け付ける状態であることがわかれば，I/Oポートにデータを出力します．

8086 CPU

●汎用レジスタ群●

15 AX 0 アキュムレータ
7 AH 0 7 AL 0

BX ベースレジスタ
BH BL

CX カウントレジスタ
CH CL

DX データレジスタ
DH DL

●インデックスレジスタ群●

15 SI 0 ソースインデックス

DI ディスティネーションインデックス

●特殊レジスタ群●

15 BP 0 ベースポインタ

SP スタックポインタ

IP インストラクションポインタ

●セグメントレジスタ群●

15 CS 0 コードセグメント

DS データセグメント

ES エクストラセグメント

SS スタックセグメント

●フラグレジスタ●

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OF DF IF TF SF ZF AF PF CF

メモリ空間
1Mバイト

マシン語命令

I/Oポート空間
64Kバイト

ディスクドライブ
プリンタ
キーボード

図 2-4　8086CPU におけるマシン語命令の世界

このように，マシン語命令の世界でハードウェアをコントロールするには非常に多くのステップを必要とし，手順を１つ間違えただけでもうまくいきません．そのかわり，ハードウェア的に実現できる機能をあますところなく利用することができます．たとえば，ディスク装置ならばハードウェア的に実現可能なあらゆる特殊なフォーマットを扱うことが可能です*．

第二に，メモリやI/Oポートの操作だけですべての機能が実現できるだけに，プログラムミスは予想外の危険な結果を持たらします．アドレス自身やアドレスを算出する手順を間違えたりすると，予定外のメモリやI/Oポートにアクセスすることになります．へたをすると，プログラム領域に書き込みを行ってしまったり，関係のない周辺機器に指令を送ってしまうことにもなりかねません．プログラムが暴走してしまったり，大事なディスク上のデータを破壊するという事態が簡単に発生してしまうのです．

アセンブラでのプログラム作成では，ごく簡単なミスから重大なバグを発生させることも多いので，十分慎重な態度でプログラミングに臨んでください．

*ディスクのフォーマットを一部変えるのはコピープロテクトの常套手段である．

COLUMN

ハードウェア情報の入手方法

VRAMがどのアドレスのメモリ領域に割り当てられているか，どのI/Oポートにどの周辺機器が接続されているかは，各機種によって異なります．また，同じ周辺機器でも，機種によってそれぞれ制御の方法が異なります．実際のマシン語プログラムで周辺機器を制御するためには，自分の機種ではその仕組みが具体的にどうなっているのかを知らなければなりません．

メーカーによっては，「ハードウェアマニュアル」がマシンに付属，あるいは別売で提供されています．これは，そのマシンのハードウェアの仕組みや制御の方法を解説したものです．「ハードウェアマニュアル」が提供されない機種や，そこに記載されている情報が十分でない機種もあります．このためメーカー以外の出版社からも「テクニカルデータブック」，「システム解析」などのかたちでハードウェアに関する資料が提供されています．これらの資料は，アセンブラで本格的にプログラムを作成する際に必須のものですから，必要に応じてそろえておくとよいでしょう．

また，周辺機器の制御に，広く一般的に使われているLSIが採用されている場合には，I/Oポートのアドレスだけが記載され，くわしい情報が省略されていることもあります．この場合は，そのLSIのマニュアルや，それを利用したプログラム集などを参照するとよいと思います．

ソフトウェア環境

ここまでの解説を読んだだけでは，アセンブラでプログラムを作成することは，非常にたいへんなことのように思えるかもしれません．事実，前節で解説した裸のハードウェア環境だけを考えた場合には，プログラムの作成に非常に多くの労力を必要とします．しかしその労力は，アセンブラをとりまくもう1つの世界である，ソフトウェア環境によって大幅に軽減されます．

その1つはROM　BIOSと呼ばれるサブルーチン群であり，もう1つはMS-DOSのシステムコールです．

ROM BIOS

MS-DOSマシンの多くは，ROMエリアのメモリに各種の周辺機器とやりとりするための基本的な入出力プログラムが格納されています．そこには，たくさんのプログラムがサブルーチンとして他のプログラムから呼び出せるような形で用意されています．これらのサブルーチン群を総称して，ROM（ロム）BIOS（バイオス）(Basic Input Output System)と呼びます．

ROM　BIOSのなかのサブルーチンは，ハードウェアの機能にそのまま対応した入出力を行うルーチンです．ハードウェアを操作するプログラムは多くのステップを必要とすることはすでに解説しましたが，ROM　BIOSを呼び出すことにより簡単に実現することができます．数個のパラメータをレジスタやメモリにセットし，ソフトウェア割り込みを使ってROM　BIOSを呼び出すだけでよいのです．

ROM BIOSに用意されている機能の内容や，ROM BIOSを呼び出す方法は機種によって異なりますが，一般的に次ページの図2-5のような手順で呼び出します．

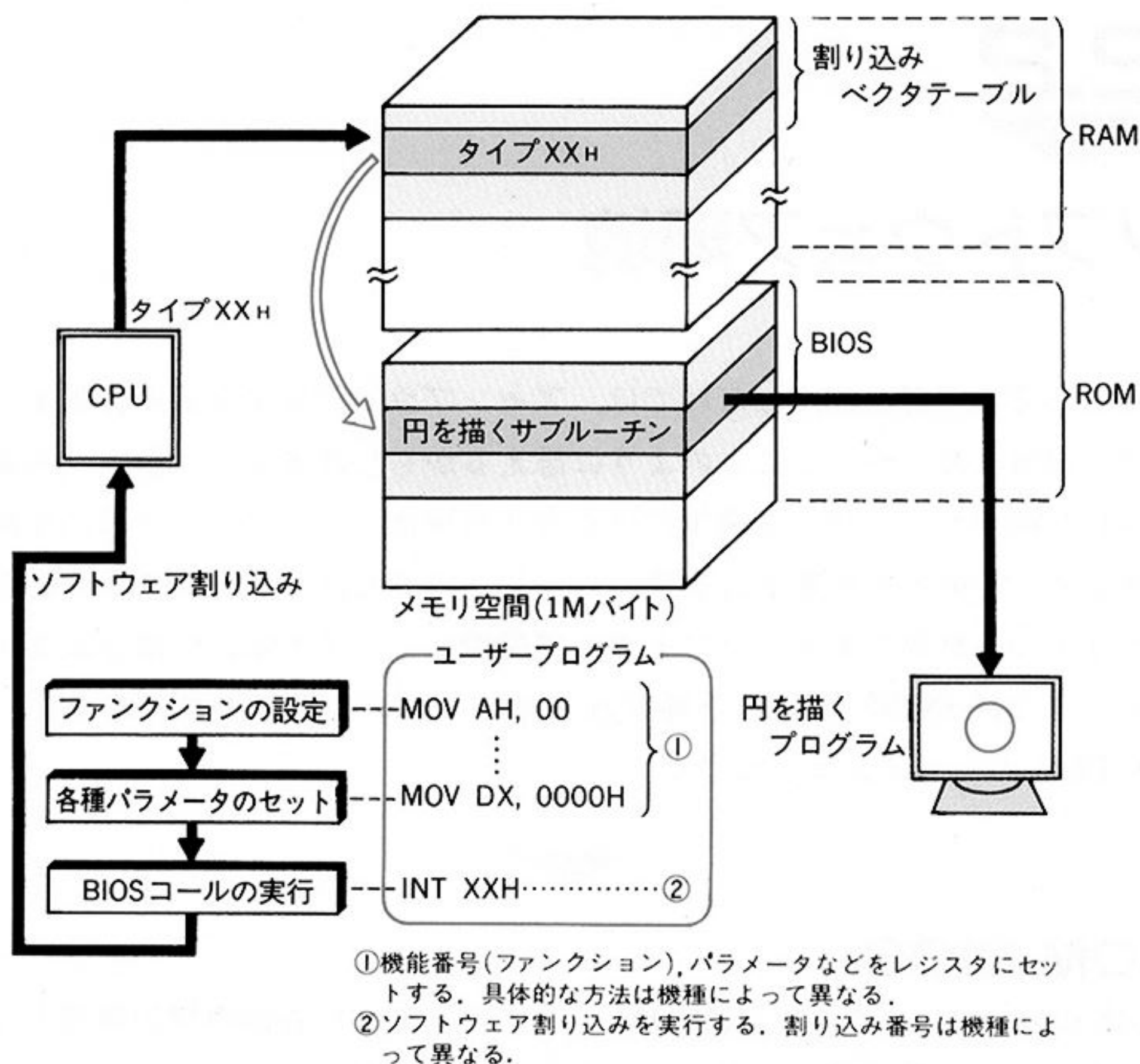



図 2-5　ROM BIOSの呼び出し手順

技術の進歩により新しい周辺機器が開発されるなど，ハードウェアの構成は年々複雑になっています．そして，それらをコントロールするプログラムを作成するためには，より多くの知識を必要とします．したがって，マシンの性能を十分に引き出すには，ROM BIOSの機能を利用せざるをえません．

機種ごとに固有の機能を持つROM BIOSは，ハードウェアの構成とともにマシンの性能を大きく左右するものです．なぜなら，どんなに高機能なハードウェアでも，ROM BIOSがそれを生かすようにできていなければ，活用することは難しいからです．このことから，ハードウェアとROM BIOSを含めたマシンの機能全体を，本書では**ファームウェア***と呼ぶことにします．

---

*ファームウェアとは，本来はCPUのチップ内にあるマシン語命令を解釈するプログラムのことを意味する．

ファームウェアは，マシンが自ら持っているハードウェア環境とソフトウェア環境であり，そのマシンで利用できる固有の世界です．ここで，ファームウェアは，同一の機種にだけ共通の世界であることに注目しておいてください．

ハードウェアの違いを吸収するMS-DOS

機種が違えばファームウェアも異なります．たとえCPUやメモリ構成が同じでも，機種ごとに独自の世界を持っているわけです．ところが，私たちが使っているアセンブラ(MASM)は，MS-DOSの動作するマシンならばどの機種でも動作します．ファームウェアが異なるはずの他機種でも同じプログラムが動作するのはどういうわけでしょうか．

この秘密はいうまでもなくMS-DOSにあります．MS-DOSは，ファームウェアの違いを吸収して共通の世界を作り出す役割を持っているのです．MS-DOSの役割をわかりやすく示したのが図2-6です．図ではMS-DOS

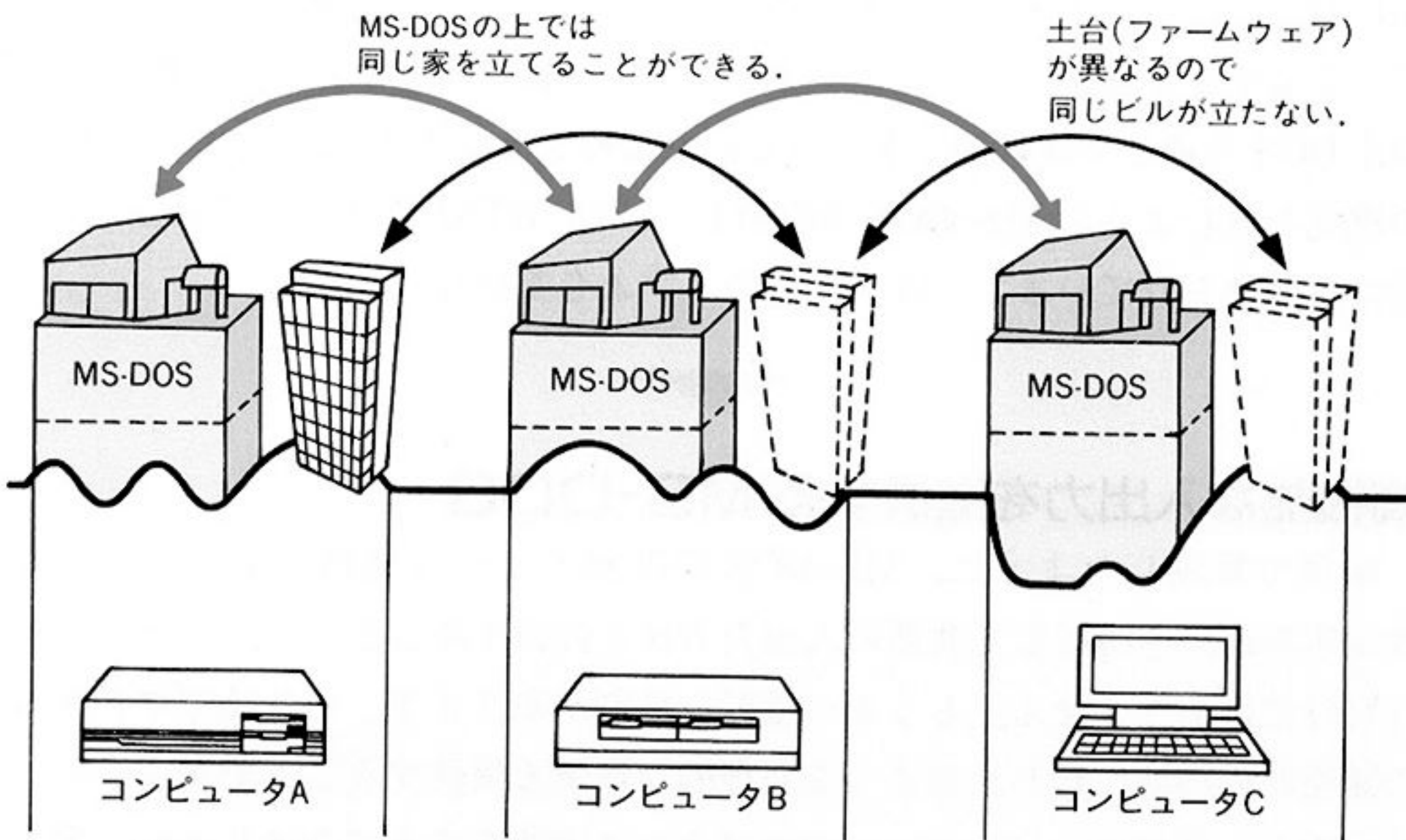



図2-6　MS-DOSの役割1

の世界を地面と建物にたとえてあります．各機種のファームウェアにあたる地面は，いろいろな個性を持ちそれぞれ地形が異なります．MS-DOS は地形(ファームウェア)の上に造成した土台のようなものです．

この土台(MS-DOS)の上に立ってみると，どの土地(機種)でも同じ地形をしています．土台(MS-DOS)の上ならば，どこでも同じ建物(プログラム)を建てることができるのです．逆に地面(ファームウェア)の上に直接建物(プログラム)を建ててしまうと，他の土地(機種)に同じ建物を建てることはできません．設計をやりなおさなければならないからです．

プログラムは，ROM BIOS を呼び出すのと同じように，MS-DOS をサブルーチンとして呼び出すことによって入出力を行うことができます．MS-DOS はプログラムからの指令を受け取ると，各機種に固有の ROM BIOS を呼び出すなどの方法でファームウェアにアクセスし，入出力を実行します．プログラムは MS-DOS を呼び出すことにより，ファームウェアに依存することなくどの機種でもまったく同じ方法で入出力を行うことができるのです．

ただし，MS-DOS は開発された時代に一般的であったハードウェアを対象に設計されたシステムであるため，ファームウェアの機能の一部を吸収しているにすぎません．たとえばグラフィックをはじめとする多くの機能が，MS-DOS を通じては利用できないという制約があります．このため，それらの機能を含むように MS-DOS を改良した MS-WINDOWS や MS-OS/2 が新たに開発されています(50 ページのコラムを参照)．

高機能な入出力を提供する MS-DOS

前項で解説したように，MS-DOS の役割の 1 つは機種によるファームウェアの違いを吸収して共通の入出力方法を提供することです．しかし，それだけではありません．もう 1 つ重要な役割があります．それはプログラムの開発が効率的に行われるような合理的な環境を提供することです．

一例としてフロッピーディスクのアクセス方法を考えてみましょう．ディスク上のデータは，ハードウェア的には，セクタという単位で読み書きします．1 セクタに格納されるデータの量は，ディスクの種類によって固定的に

決められています．たとえば国産機種の場合，1Mバイトディスク(2HD)の1セクタは1024バイト，640Kバイトディスク(2DD)の1セクタは512バイトです．1枚のディスクにはたくさんのセクタが存在し，それぞれのセクタは物理的な位置を表すトラック番号とセクタ番号によって区別します．ディスクからデータを読み込む際には，必ず1セクタ分ずつのデータを読み込むことになります．

ROM BIOSはハードウェアの機能にそのまま対応した形での入出力を提供しています．すなわち，図2-7-bのようにトラック番号，セクタ番号などの物理的な位置を指定して呼び出すことにより，1セクタ分のデータを読み出したり書き込んだりすることができます．データを書き込んだ場所を覚えておくためには，トラック番号とセクタ番号を記録しておかなければなりません．

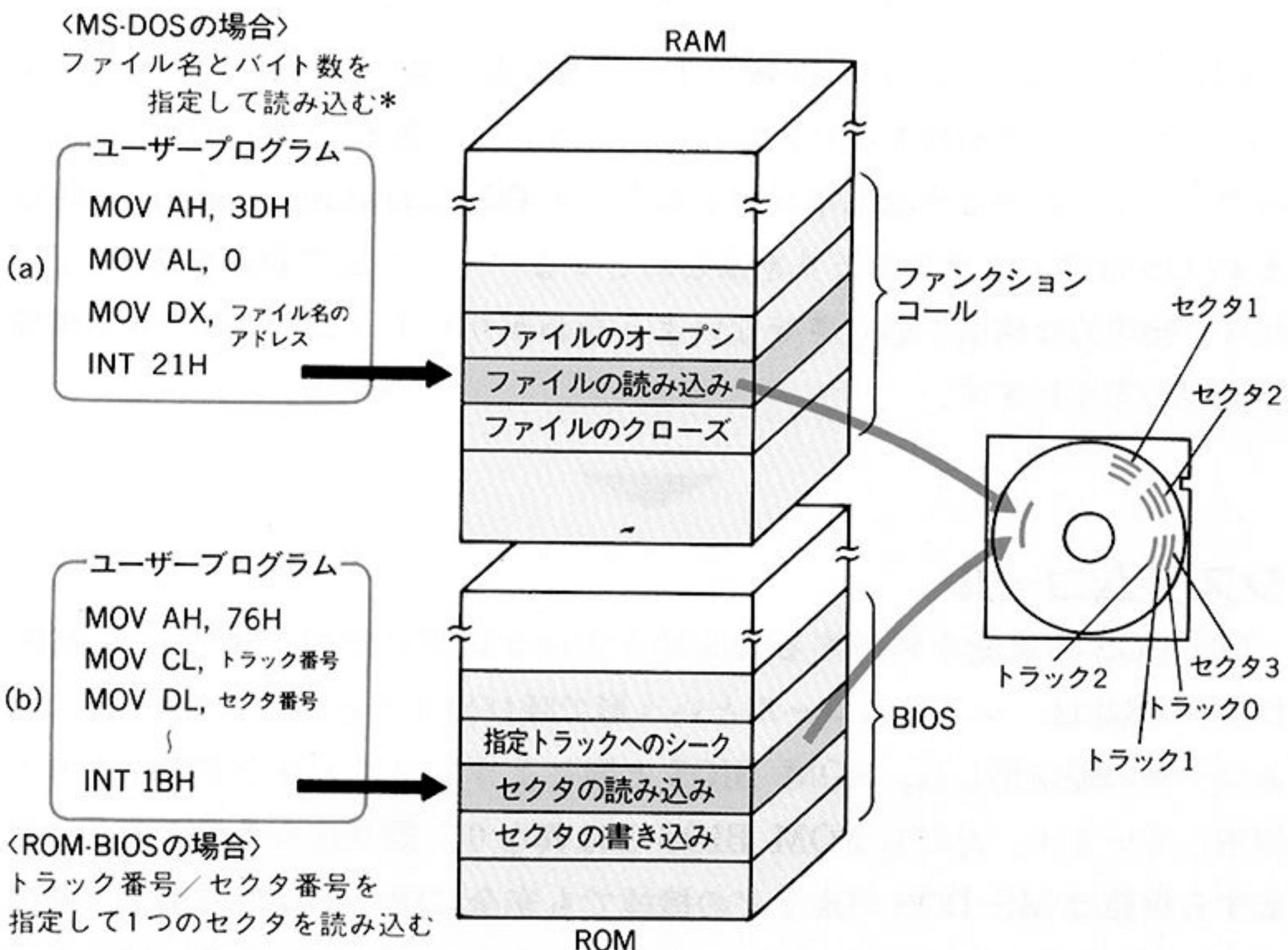



図2-7　MS-DOSの役割2

これに対し，MS-DOSではディスク上のデータをファイルという単位で扱います．1つ1つのファイルは，ファイル名によって区別します．また，ファイルの大きさは格納するデータの量に応じて任意に変化させることができます．図2-7-aのようにファイル名と読み込むバイト数を指定してMS-DOSを呼び出すことにより，任意の量のデータを読み出したり書き込んだりすることができます．データを書き込んだ場所を覚えておくためには，ファイル名を記録しておくだけでよいのです．

このことからわかるように，MS-DOSではハードウェアの機械的な仕組みを意識することなくディスク上のデータを読み書きすることができます．ファイルとして書き込んだデータの格納位置や，ディスクの種類による違いはMS-DOSがうまく管理してくれるので，単純な統一された方法でデータを保存することができます．ディスク装置との入出力に関する面倒な処理はMS-DOSにまかせて，データの処理そのものに専念することができるわけです．

MS-DOSのように，いわば縁の下の力持ち的な働きをするソフトウェアは，プログラムが動作するためになくてはならない基本ソフトといってもよいでしょう．このような基本ソフトのことをOS(Operating System)と呼びます．OSはディスクやメモリをはじめとするハードウェア資源を管理し，それらを物理的な構造を感じさせないようなわかりやすい形で利用できる環境を提供してくれます．

システムコール

MS-DOSの機能を利用する具体的な方法を以下に解説しましょう．MS-DOSの機能は，**システムコール**という形で呼び出すことができます．システムコールの呼び出しは，ROM BIOSと同じようにソフトウェア割り込みを利用しています．ただしROM BIOSとは異なり，割り込み番号とそれに対応する機能はMS-DOSの走るどの機種でも完全に統一されています．そのためMS-DOSの世界を見る限りどの機種でもまったく同じ世界に見えるのです．主なシステムコールの割り込み番号とその機能を**表2-1**に挙げておきます．

割込み番号	機　　能
INT 20H	プログラムの終了
INT 21H	ファンクションコール
INT 25H	物理セクタ番号によるディスク読出し
INT 26H	物理セクタ番号によるディスク書込み
INT 27H	プログラムの常駐終了

表 2-1　主なシステムコール

表 2-1 のなかで，割り込み番号 21_H のシステムコールは，特に**ファンクションコール**と呼ばれています．ファンクションコールは MS-DOS の機能を呼び出すための最も一般的な手段であり，これ以外のシステムコールを利用することはあまりありません*．

*たとえば，DISKCOPY コマンドは割り込み番号 25_H，26_H のシステムコールを利用している．これはディスクの内容をファイル単位ではなく，物理的な構造のまま，まるごとコピーするという特殊な目的のためである．

ファンクションコールで利用できる機能(ファンクション)は全部で約70種類あります*．1つ1つの機能には，すべて番号が付けられています．ファンクションコールではその番号を指定して呼び出すことにより，機能を選択します．このファンクションコールを呼び出す方法を図解したのが図2-8です．また，本書のプログラムで利用しているファンクションを表2-2に示します．

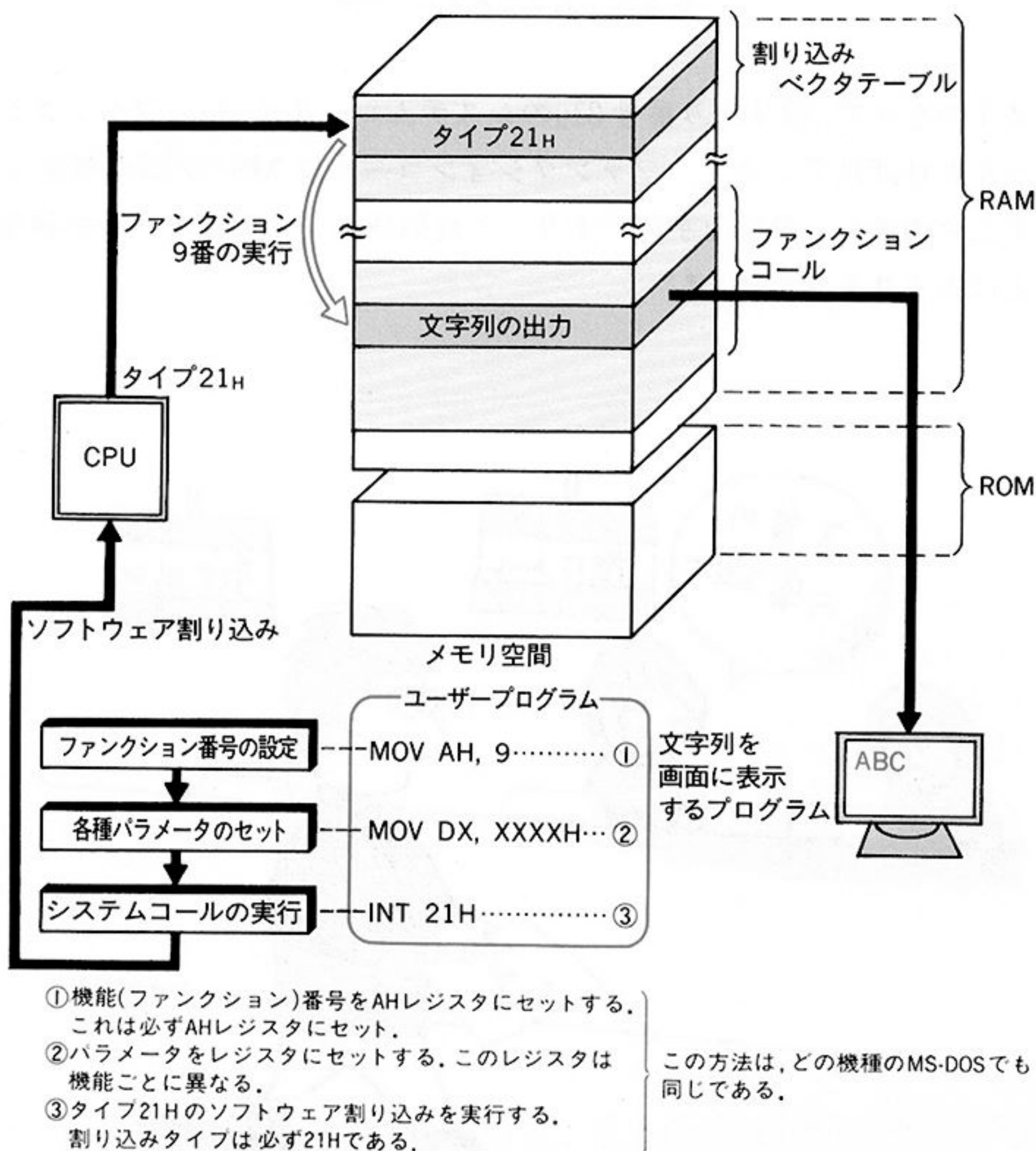
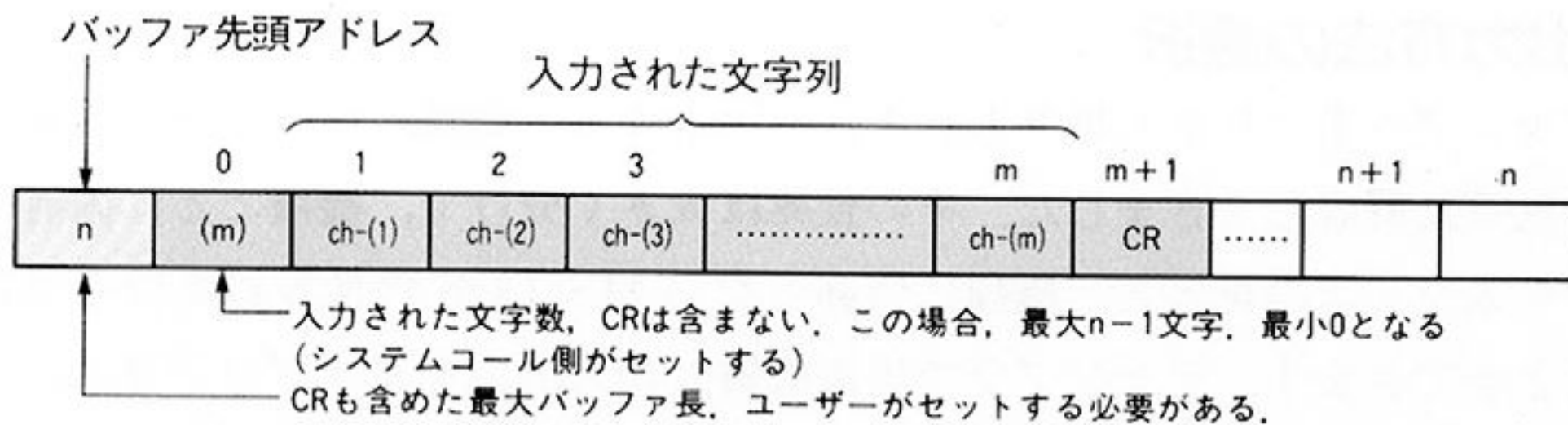



図2-8　ファンクションコールの仕組み

---

* MS-DOS　Ver2.11の場合．Ver3.1では約100種類ある．全ファンクションコールの詳細は，APPENDIXを参照のこと．

番号	機　能	呼び出し方法	実行結果
02_H	文字出力 (^Cチェックあり)	MOV　AH,02H MOV　DL,標準出力に出力する文字コード INT　21H	なし
06_H	コンソール 直接入出力 (エコーなし 入力を待たない ^Cチェックなし)	入力 MOV　AH,06H MOV　DL,0FFH INT　21H	ゼロフラグがセットされたとき AL←00_H ゼロフラグがリセットされたとき AL←入力された文字コード
		出力 MOV　AH,06H MOV　DL,標準出力に出力する文字コード (0FFH以外) INT　21H	なし
08_H	エコーなし キーボード入力	MOV　AH,08H INT　21H	AL←標準入力から入力された文字コード
09_H	文字列の出力 (^Cチェックあり)	MOV　AH,09H MOV　DX,標準出力に出力する文字列の先頭アドレス* (文字列の終わりは'$'で識別する) INT　21H	なし
$0A_H$	バッファード キーボード入力 (^Cチェックあり)	MOV　AH,0AH MOV　DX,キーボード入力バッファの先頭アドレス* INT　21H	なし
25_H	割り込み ベクタの設定	MOV　AH,25H MOV　DX,割り込み処理ルーチンのエントリアドレス* INT　21H	なし
31_H	プログラムの 常駐終了	MOV　AH,31H MOV　AL,リターンコード MOV　DX,メモリに常駐するプログラムのパラグラフサイズ** INT　21H	なし
35_H	割り込み ベクタの読出し	MOV　AH,35H MOV　AL,割り込みベクタ番号 INT　21H	ES:BX←割り込みベクタ

●キーボード入力バッファ

番号	機　能	呼び出し方法	実行結果
$3F_H$	ファイル／デバイスの読み出し	MOV　AH,3FH MOV　DX,ディスク入力バッファの先頭アドレス* MOV　CX,読み込むバイト数 MOV　BX,ファイルハンドル IVT　21H	キャリーがリセットされた場合 AX←読み込まれたバイト数 キャリーがセットされた場合 $AX=05_H$　指定されたファイル・ハンドルは読み出しが許可されていない $AX=06_H$　指定されたファイル・ハンドルはオープンされていない
40_H	ファイル／デバイスの書き出し	MOV　AH,40H MOV　DX,ディスク出力バッファの先頭アドレス* MOV　CX,書き出すバイト数 MOV　BX,ファイルハンドル INT　21H	キャリーがリセットされた場合 AX←書き出されたバイト数 キャリーがリセットされた場合 $AX=05_H$　指定されたファイル・ハンドルは書き込みが許可されていない $AX=06_H$　指定されたファイル・ハンドルはオープンされていない
$4C_H$	プロセスの終了	MOV　AH,4CH MOV　AL,リターンコード INT　21H	

＊印の付いたものは，DSレジスタにそのセグメントアドレスがセットされていなければならない．
（COMモデルでは必ずそうなっている）
＊＊パラグラフサイズについては，4章124ページで解説する．

表 2-2　本書のプログラムで利用したファンクションコール

1章で紹介したプログラムにもファンクションコールを利用しています．次ページの図2-9にその部分を示します．アセンブラでプログラムを組む場合には，このようにシステムコールの恩恵にあずかることがほとんどです．

入出力方法の選択

アセンブラをとりまく世界として，ハードウェア環境，ソフトウェア環境をそれぞれ解説してきました．その世界は大きく分けて，機種ごとに固有のファームウェアの世界と，機種に依存しないMS-DOSの世界の2つを考えることができます．アセンブラで周辺機器との入出力を行うプログラム，すなわちハードウェアを操作するプログラムを作成する場合には，どちらの世界を相手にするかを選ばなければなりません．

```
        ASSUME  CS:CODE,DS:CODE
CODE    SEGMENT

        ORG     100H
START:  MOV     BX,0
NOINPUT:MOV     AH,06H
        MOV     DL,0FFH             コンソールからの直接入力
        INT     21H
        JNZ     PRINT
        INC     BX
        CMP     BX,5
        JGE     START
        JMP     NOINPUT
PRINT:  SHL     BX,1
        MOV     DX,TABLE[BX]
        MOV     AH,09H              文字列出力
        INT     21H
        MOV     AH,4CH
        MOV     AL,00H              プロセスの終了
        INT     21H

HARE    DB      'HARE',0DH,0AH,'$'
KUMORI  DB      'KUMORI',0DH,0AH,'$'
AME     DB      'AME',0DH,0AH,'$'
ANOTI   DB      'AME NOTI HARE',0DH,0AH,'$'
KNOTI   DB      'KUMORI NOTI AME',0DH,0AH,'$'

TABLE   DW      OFFSET HARE,OFFSET KUMORI,OFFSET AME
        DW      OFFSET ANOTI,OFFSET KNOTI

CODE    ENDS
        END     START
```

図2-9 ファンクションコールを利用したプログラム例

ファームウェアの世界，MS-DOSの世界のそれぞれについて特徴をまとめておきましょう．

MS-DOSの世界における入出力

前節で解説したように，MS-DOSを介して入出力を行うようにすれば，ハードウェアの機械的な仕組みを意識することなく，しかもMS-DOSの動作するマシンならばどの機種でも共通に動作するプログラムを作成することができます．

しかし，MS-DOSはDOS(Disk Operating System)という名が示すよう

にファイル管理に関してはかなり豊富な機能を持っていますが，それ以外の点については十分とはいえません．ファームウェアの機能のすべてをサポートしているわけではないのです．

たとえば，キャラクタ画面の制御の方法としては，カーソルの移動や画面のクリアなどの簡単なエスケープシーケンスしか用意されていません*．エスケープシーケンスを利用して画面制御を行うと，MS-DOSの内部処理に時間がかかり，満足な表示速度が得られない場合があります．複雑な画面操作などもできません．また，グラフィックに関しては，MS-DOSではまったくサポートされていません．最近のパソコンは高度なグラフィック機能を持っているにも関わらず，MS-DOSの世界ではそれを利用する手段がないのです**．このように，MS-DOSレベルでは，機種の壁を超えた互換性を得られる代わりに，ハードウェアの性能を最大限に利用することはできません．

ファームウェアの世界における入出力

MS-DOSの世界で利用できないようなハードウェアの性能を引き出すためには，MS-DOSを介さずにファームウェアの世界を直接利用することになります．たとえば，エディタなどで非常に高速なスクロールを売り物にしているものがありますが，これらはVRAMを直接アクセスしています．MS-DOSを介してエスケープシーケンスによってスクロールを行うよりも高速に，しかもエスケープシーケンスでは不可能な部分的なスクロールなどが簡単に実現できます．グラフィック機能を利用するプログラムもVRAMやROM BIOSなどのファームウェアを利用しています．

MS-DOSの世界を利用したプログラム，ファームウェアの世界を利用したプログラムは，前節の建物のたとえでいえば，図2-10のような形をしています．ファームウェアの世界を利用した建物（プログラム）は土台を突き抜けて地面にまで達しています．これはMS-DOSの機能を利用するだけでなく，ファームウェアをも直接利用していることを表しています．

---

*エスケープシーケンスについては48ページのコラムで解説する．

** MS-DOSでは，あとから追加したハードウェアや機種に固有のハードウェアを，デバイスドライバを用意することによってシステムに組み込むことができる．そうすればMS-DOSを介して入出力を行うことができるが，機種ごとに同様のデバイスドライバを作成しなければ，結局は機種に依存した方法となってしまう．

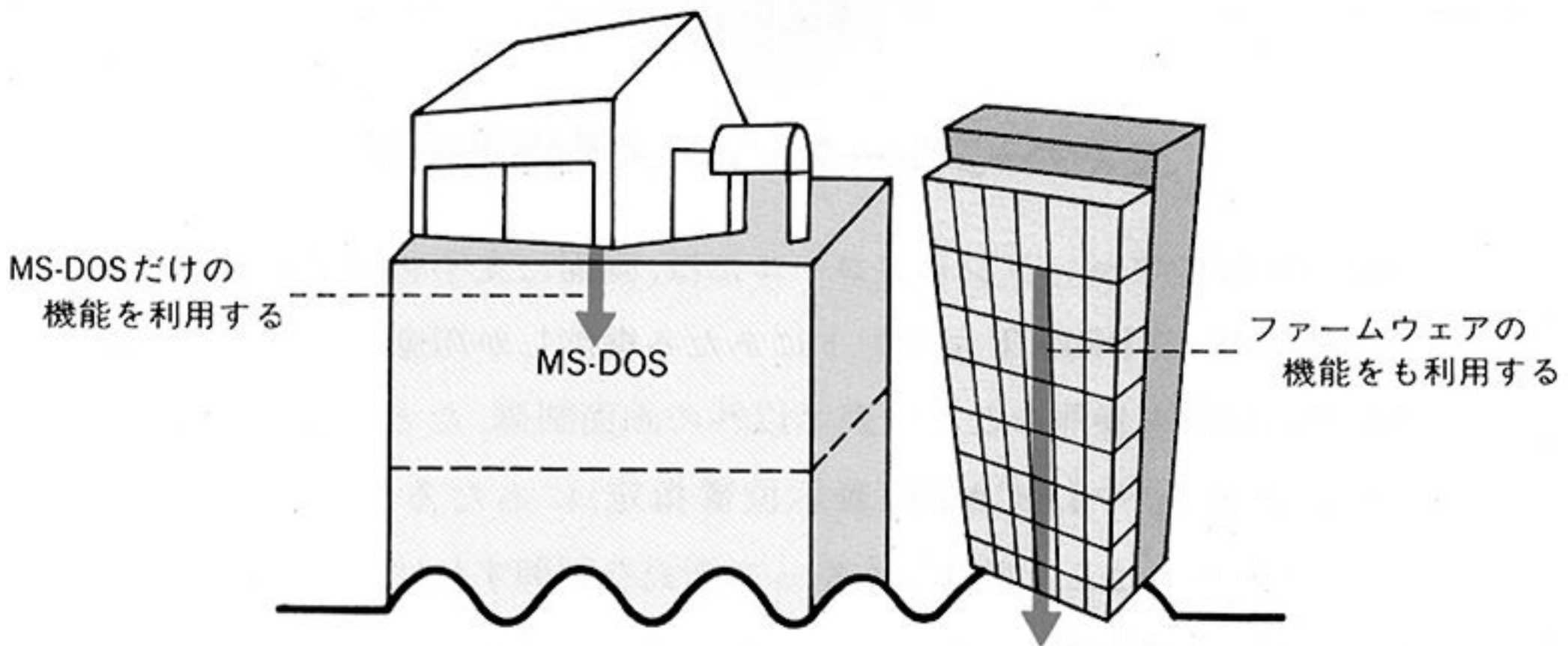



図 2-10 ファームウェアの世界まで利用するプログラム

また，ファームウェアの世界まで利用したプログラムは，図 2-10 でわかるように機種に依存したプログラムになってしまいます．機種によらない環境を提供する MS-DOS の主旨からすれば，不法なプログラムであるともいえます．しかし，MS-DOS がハードウェアの進歩に対応できず十分な機能を提供できない限り，しかたのないことでしょう．ハードウェアがせっかく持っている機能を活用しない手はありません．

ただし注意しなければならないことが 1 つあります．それは MS-DOS の世界を逸脱してファームウェアをアクセスすることは，危険を伴っているということです．特に MS-DOS がサポートしているデバイス(たとえばキーボードなど)を MS-DOS を通さずにアクセスすると，MS-DOS と衝突してハングアップしたり，システムを破壊する可能性もあります．ファームウェアの世界のアクセスは，その点に十分気をつけなければなりません．

ファームウェアの世界まで利用したプログラムでも，ディスクの入出力については MS-DOS の世界を利用するのが一般的です．したがって，そのようなプログラムで作成したファイルでも他の機種で読み込むことが可能です．たとえば，ある機種のエディタで作成したファイルを他機種のエディタで編集することができます．このことは MS-DOS の大きな利点の 1 つであり，MS-DOS の OS としての存在意義もそこにあるといえます．

COLUMN

エスケープシーケンスによる画面制御

MS-DOSのファンクションコールには，画面に文字を表示する方法として，BASICのPRINTコマンドにあたる機能しか用意されていません(APPENDIXを参照のこと)．表示以外の画面制御，たとえばBASICのCLS(画面消去)やLOCATE(表示位置指定)にあたるファンクションコールはありません．ではどうやって画面を制御すればよいのでしょうか．その答えがエスケープシーケンスです．

エスケープシーケンスは，画面制御のコマンドを特殊な文字列で表します．その文字列をファンクションコールを使って「表示」すると，その文字列自身が表示される代わりに，画面消去などの画面制御が実行されます．その特殊な文字列とは，本当に表示すべき文字列と区別するためにエスケープコードという文字で始まることになっています．エスケープコードで始まる文字列なので，エスケープシーケンスと呼びます．この仕組みと主なエスケープシーケンスを次に示しましょう．

たとえば，下の図に示すようなコマンドで画面を制御することができますが，エスケープシーケンスはMS-DOSの仕様として厳密に決められているわけではありません．推奨されているシーケンスもありますが，必ずしもすべての機種がそのシーケンスを採用しているわけではありません．むしろ機種によってシーケンスが微妙に異なるのが普通です．

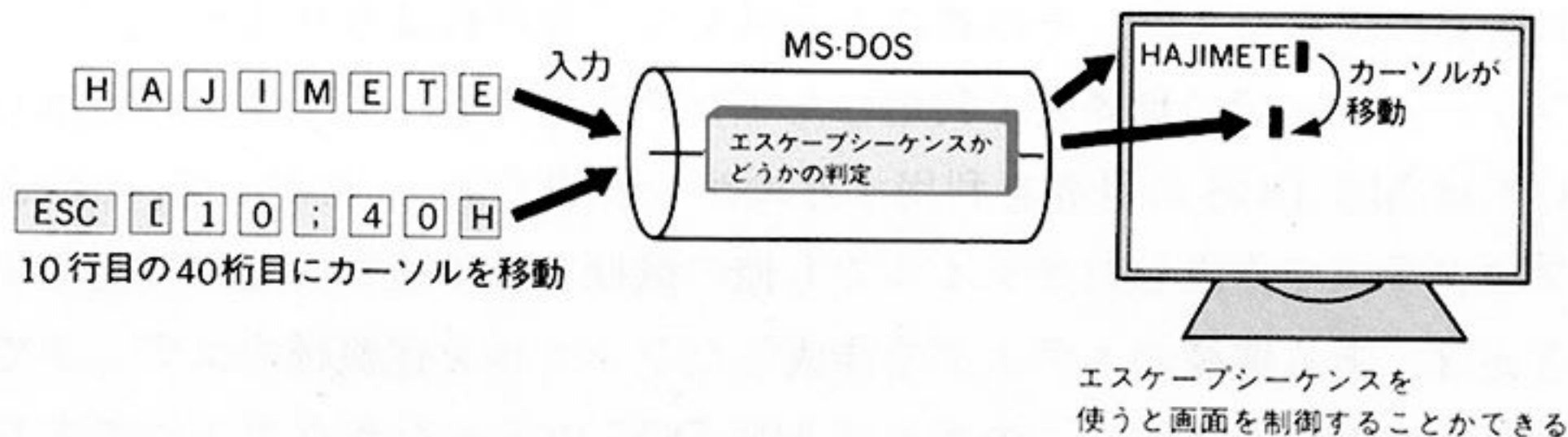



図　エスケープシーケンスの仕組み

フォーマット	解　　説
ESC [pl;pcH	カーソルを，pl行目のpc桁目に移動する
ESC [2J	画面をクリアし，カーソルをホームに移動する
ESC [ps;……;ps m	psで指定した表示属性をセットする．これ以降に表示される文字はこの属性にしたがって表示される.psは任意個数指定できるが,機種によっては同時に指定しても無効なものがあるので注意

ps	属　　性
0	メーカーの規定した初期値
1	強調(ハイライト，もしくは太字)
4	下線付き
5	ブリンク(点滅)
7	リバース(反転)
8	シークレット(文字を見せない)
30	黒
31	赤
32	緑
33	黄色
34	青
35	マゼンダ
36	水色
37	白
40	背景　黒
41	背景　赤
42	背景　緑
43	背景　黄色
44	背景　青
45	背景　マゼンダ
46	背景　水色
47	背景　白

表　主なエスケープシーケンス

エスケープシーケンスは，もともと通信端末機で画面を制御するために考えられた方法です．端末に表示される文字は回線を通して送られてきますが，その文字のなかに画面制御のコマンドを混ぜて送ることが可能になります．MS-DOSマシンの多くは，通信端末としても利用することができるようになっており，各メーカーは自社の大型コンピュータの端末で採用しているエスケープシーケンスをそのまま採用している場合が多いのです．

COLUMN

MS-Windows, OS/2のマルチウィンドウ環境

MS-DOSの後継OSである，MS-WindowsやMS-OS/2は，MS-DOSの機能を大幅に拡張したものとなっています．特にグラフィックやマウスといった比較的新しい周辺機器をシステムの機能としてサポートしている点が注目されます*．

本文で解説したように，MS-DOSではこれらの機能がサポートされておらず，アプリケーションプログラムが個々に対応していました．このためアプリケーションによって操作方法が違ったり，他機種では利用できなかったりするという問題がありました．

メモリの容量やスピードの問題からあまり普及はしていないMS-Windowsですが，OSによりグラフィックをサポートするという方向に期待が持たれています．MS-Windowsでは，グラフィックを使用したプログラムでも機種によらずに動作するからです．たとえば，アメリカのIBM-PCのMS-Windows用に作られたプログラムはほとんどの場合，国産機種のMS-Windows上でそのまま動きます．

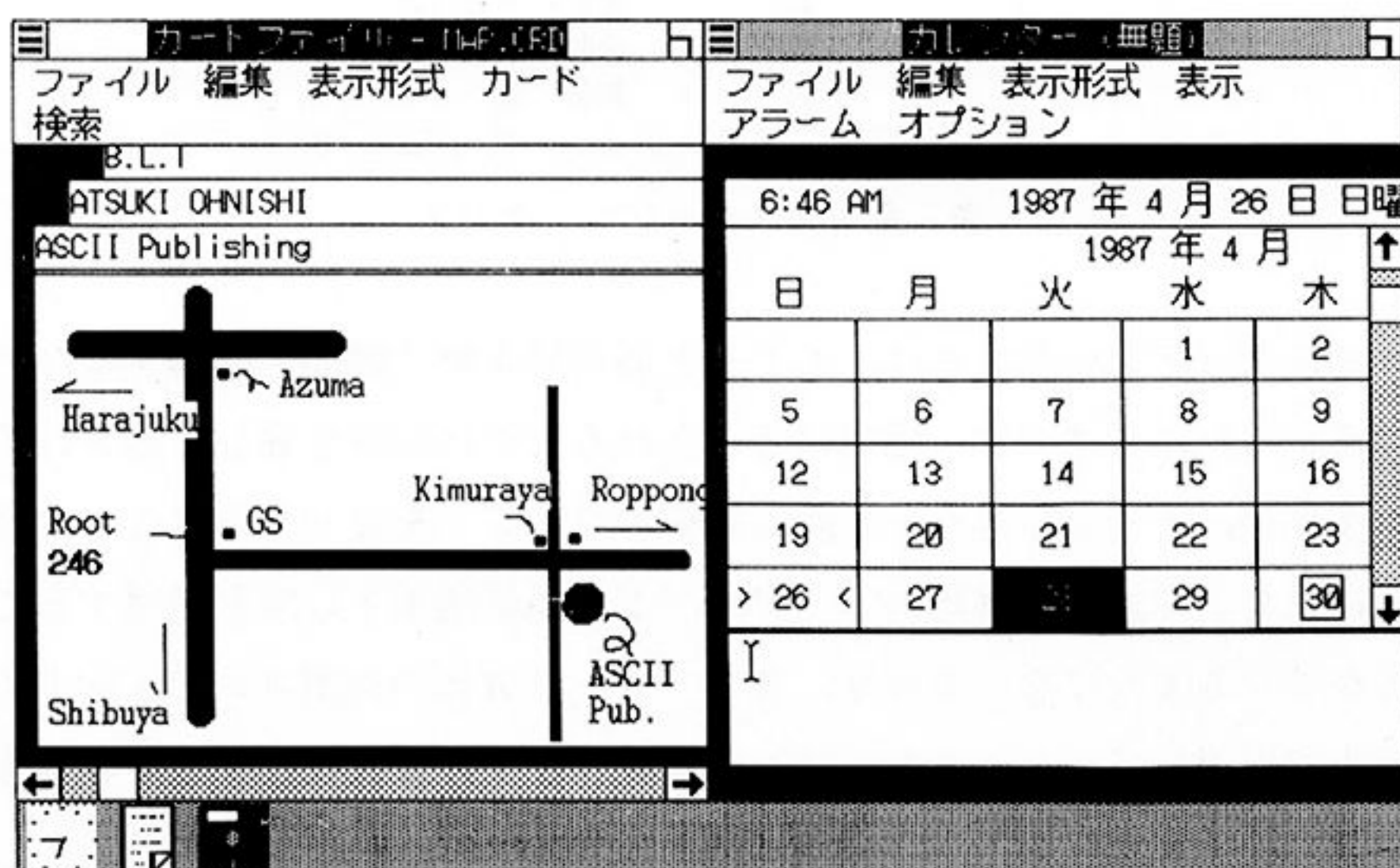



図　MS-WINDOWS(Ver1.0)

MS-Windowsは，その名のとおりMS-DOSにウィンドウの機能を取り込んだものです．図のように画面をウィンドウ(窓)と呼ばれる小画面で区切って，そのなかを1つの画面であるかのように扱います．もちろん文字だけでなく，グラフィックも表示することができます．さらに，いくつものウィンドウを開いてそれぞれにアプリケーションを割り当てることができます．たとえば表計算のウィンドウとワープロのウィンドウを開いておいて，切り替えながら使うことが可能です．

また，マウスを利用した使いやすい操作環境も提供されています．たとえばマウスを使って図形を描いたり，画面上のボタンやプルダウンメニューを操作することにより，アプリケーションに指示を与えることができます．マウスで指定した範囲のデータを移動／複写したり，さらに他のアプリケーションへ複写することも可能です．これを実現する仕組みは，システムによりサポートされているので，どのアプリケーションでも同様の操作方法で，これらの機能を利用することができるのが大きな特徴になっています．

なお，OS/2でもMS-Windowsと同様の環境が提供されます*．そして，OS/2にはさらに注目すべき機能が数多く追加されています．くわしくは8086CPUの上位バージョンである80286CPUの解説とあわせてAPPENDIXで紹介します．

---

* OS/2ではバージョン1.1からプレゼンテーションマネージャとしてマルチウィンドウ環境がサポートされる．

2.3 プログラミング環境としてのアセンブラ

アセンブラをとりまく世界についての理解が深まったところで，アセンブラでプログラムを作成することの意義，アセンブラで作成するのにふさわしいプログラムとはどのようなものかを解説しておきます．

アセンブラの世界でのプログラミング

これまで，マシン語命令の世界，ファームウェアの世界，MS-DOSの世界，といったハードウェア，ソフトウェア両面にわたるアセンブラをとりまく世界について解説してきました．アセンブラの世界でのプログラミングがどのようなものかはだいたいつかめたのではないかと思います．

アセンブラでのプログラミングを理解するということは，コンピュータ自身の仕組みやソフトウェアが動作する仕組みを理解することにつながります．高級言語でしかプログラムを作成しないユーザーであっても，その背景にある世界を知ることで，ただの箱であったコンピュータを身近なものに感じることができるでしょう．

しかし，最近になってハードウェアの性能の進歩にともない，アセンブラはほんの一部のプログラムでしか使われなくなってしまいました．ほとんどの部分は高級言語で記述されます．それはなぜでしょうか．

１つにはアセンブラの世界，すなわちマシン語の世界でのプログラミングが高級言語にくらべてわかりにくい，ということが挙げられます．それは，プログラムの最小単位である，１つ１つのマシン語命令が非常に単純な機能しか持っていないからです．これに対して，高級言語はアセンブリ言語のプログラムのわかりにくさに起因する生産効率の低さをカバーし，純粋にアルゴリズムのみをプログラムとして記述する目的で開発されました．とくに最近多くの分野で使われているＣ言語は，高級言語でありながらアセンブラ並

みの細かい記述ができ，しかもできあがった実行ファイルはコンパクトで，実行速度が速いという優れた特徴を持っています．

また，アセンブラで書いたプログラムは，デバッグが難しいということもいえます．アセンブラでプログラムを作ってみれば誰しも経験することですが，スタックの操作をちょっと間違えたり，レジスタの保存を忘れただけで簡単に暴走してしまいます．暴走やおかしな動作の原因をつきとめるのは非常に困難です．なにしろ，プログラムの途中でレジスタの値などを表示させようにも，それだけで長々とプログラムを作らなければなりません．

さらに，アセンブラで書いたプログラムを他のCPUを搭載したマシンに移植するためには，全面的に書き直すしかありません．なぜなら，CPUが違えばマシン語命令の構成もまったく違うからです．これに対して，高級言語で書かれたプログラムはCPUに依存しません．

こうした理由から，アセンブラでプログラムを作ることは次第に敬遠されつつあります．今ではアセンブラは，どうしてもアセンブラでなければ，という場面に使われるにすぎません．つまり，アセンブラと高級言語の使い分けがうまくできなければならないのです．また高級言語とアセンブラを組み合わせて使おうとすると，2つのプログラム間でやりとりをする必要がでてきます．それにはコンピュータの仕組みや，アセンブラの世界でのプログラミングをよく理解しておかなければ満足するプログラムは組めません．

アセンブラの守備範囲

アセンブラに適した分野，アセンブラで書くべきプログラムとはどんなものでしょうか．主なものを挙げると次のような場合が考えられます．

1．MS-DOS，ROM BIOSを直接呼び出す
2．ハードウェアを直接操作する
3．ハードウェア割り込み処理
4．高速性を必要とするデータ処理
5．デバイスドライバの作成

それぞれについてくわしく解説しましょう．

1．MS-DOS，ROM BIOS を直接呼び出す

2.2 節で述べたように，MS-DOS のシステムコールや ROM BIOS を呼び出すには CPU のレジスタにファンクション番号やパラメータをセットしなければなりません．高級言語では通常レジスタを操作することはできないので，それは不可能です．したがって呼び出す部分だけはアセンブラで書かなければなりません．

といっても，MS-DOS 上の言語処理系には，特殊機能としてファンクションコールを呼び出す機能が用意されているものが多いようです．特に，システム記述言語とも呼ばれるC言語では，必ずといっていいほどソフトウェア割り込みを呼び出すためのライブラリが用意されています(表 2-3)．したがってC言語を使用する限り MS-DOS や ROM BIOS を呼び出すためにアセンブラを必要とすることはほとんどないでしょう．

関数名	機能
intdos( )	MS-DOSのファンクションコールを呼び出す． セグメントは指定できない．
intdosx( )	MS-DOSのファンクションコールを呼び出す． セグメントの指定が可能．
bdos( )	MS-DOSのファンクションコールを呼び出す． AX／DX レジスタを指定し，結果はAXレジスタのみが返る．
int86( )	引数で指定されたソフトウェア割り込みを実行する． セグメントは指定できない．
int86x( )	引数で指定されたソフトウェア割り込みを実行する． セグメントの指定が可能．

注：MS-C，TurboC，Lattice Cなどの標準ライブラリの場合

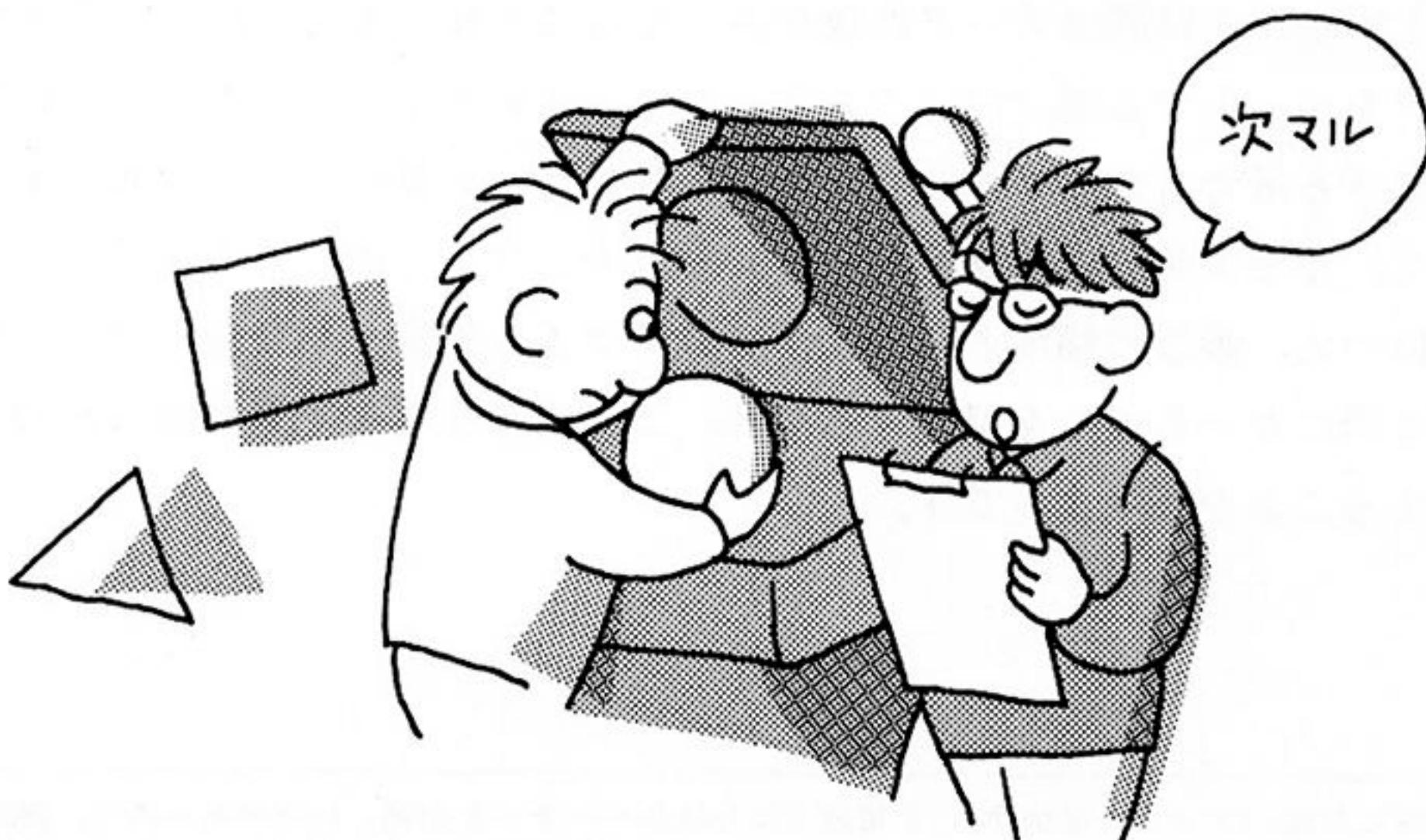
表 2-3　システムコール，ROM BIOS を呼び出すC言語の関数

2．ハードウェアを直接操作する

マシン語ではすべてのハードウェアを直接操作することができます．しかし，すでに解説したようにハードウェアを直接操作しなければならない場面はかなり限られています．MS-DOS や ROM BIOS に機能として用意されているものは，わざわざ直接操作するまでもありません．どうしても直接操作しなければならない場合でも，高級言語から操作できる場合が少なくありま

せん．高級言語にも，メモリやI/Oポートを直接操作する命令が用意されているからです．したがってどうしてもアセンブラが必要になるのは次のような場合だけでしょう．

ハードウェアを自作したり特殊な使い方をしたりする人にとっては，タイミング調整や割り込み処理などの問題から，アセンブラを使ってハードウェアを直接操作する必要があります．後で解説するように，割り込み処理はアセンブラでしか記述できません．

また，グラフィックの描画もアセンブラで直接操作する場合が少なくありません．ROM BIOSや高級言語にグラフィックを操作する機能はありますが，望む機能がすべて用意されているとは限らないからです．小さな機能を組み合わせて実現することもできますが，グラフィック画面に対する処理はデータの量が非常に多く，またビット単位の操作が必要になります．処理スピードが遅いと画面がちらついてしまったり，なめらかな動きを実現できないのです．このような問題を解決するためにアセンブラが使われます．

3．ハードウェア割り込み処理

割り込み処理は基本的にアセンブラでしか記述できません．なぜなら，ハードウェア割り込みは他のプログラムを実行中に突然発生するわけですから，割り込み処理を終了してもとのプログラムに戻るときにはすべてのレジスタの内容が元通りになっていなければならないからです．前述したように，高級言語ではレジスタを直接操作できず，また割り込み処理では，高級言語ではサポートされていない特殊なマシン語命令が必要になります．6章では，割り込みを使ったプログラム例を紹介します．このプログラムは先に解説した，グラフィックを操作する例にもなっています．

なお，アセンブラで一部を記述することにより，割り込み処理の大部分を高級言語で記述することも可能です．くわしくは他の文献を参照してください*．

4．高速性を必要とするデータ処理

ハードウェアに関連しないことでもアセンブラに適した分野があります．それはデータ処理をどうしても高速化したい場合です．大量のデータを処理しなければならない場合など，高級言語では時間のかかってしまう処理もアセンブラを使うことによって処理時間を縮めることができます．

たとえば，住所録をアイウエオ順に並べ換えるといった，データベースのソートにかかる時間はデータ件数が多くなるにつれて非常に長くなります．データをソートする部分だけでもアセンブラを利用すれば，実用的な速度で処理できる場合も少なくありません．6章では，大量のデータを処理する例として，オセロゲームの探索ルーチンをアセンブラで作ってみます．

とはいえ，最近の傾向としては，プログラムを改良したり他のマシンへ移植する際にかかる労力を軽減するため，このような場合でも高級言語のみで記述することが多いようです．

---

*「応用C言語」（アスキー出版局）にC言語で割り込みルーチンを記述した例があるので，興味のある方は参考にするとよい．また，Turbo C では interrupt 型の関数として定義することにより，割り込みルーチン全体をC言語で記述することができる．

5．デバイスドライバ

MS-DOSではデバイスドライバを組み込むことによって，システムを拡張することができます．たとえば，RAMディスクやかな漢字変換の機能をシステムに組み込むためのデバイスドライバは広く使われています．

一般にデバイスドライバは高級言語で記述することはできません．なぜなら，デバイスドライバは通常の実行型ファイルとは異なる特殊な形式でなければならず，高級言語ではその形式のオブジェクトを出力できないからです*．また，デバイスドライバは一般に新しい周辺機器などをシステムに組み込むために使われるものであり，ハードウェアを直接操作する場合が多いことも1つの理由として挙げられます．

6章では，アセンブラによるデバイスドライバの開発例を紹介します．

COLUMN

IRET, CLI, STI命令
―割り込み処理に関するマシン語命令―

割り込みルーチンは一種のサブルーチンです．ハードウェア割り込みの要求が周辺機器から発生すると，その時点で割り込みルーチンが呼び出されます．どんなプログラムを実行中でも，強制的にCALL命令が実行されることになります．ただしその際に，戻り先のアドレスに先立ってフラグレジスタの内容が自動的にスタックにPUSHされます．したがって，通常のRET命令では割り込みルーチンからもとのプログラムに戻ることはできません．RET命令はIPレジスタ(およびCSレジスタ)をスタックからPOPするだけだからです．

割り込みルーチンからもとのプログラムに戻るために専用に用意されているのがIRET(Interrupt RETurn)命令です．IRET命令では，スタックから戻り先アドレスをPOPするとともにフラグレジスタもPOPします．

---

*一部のみをアセンブラで記述することにより，高級言語でデバイスドライバの大部分を記述することは可能．くわしくは巻末に挙げた参考文献を参照のこと．

また，割り込みを一時的に禁止したい場合があります．たとえば，割り込みベクタを変更する場合がそうです．割り込みベクタはセグメントアドレスとオフセットアドレスの両方をセットしなければなりませんが，このときどちらか片方だけをセットした状態でそのベクタに対応した割り込みが発生するとどうなるでしょうか．誤ったアドレスが割り込みベクタとして扱われ暴走してしまいます．このため割り込みベクタを変更している間は割り込みを禁止しなければなりません．

8086CPUには，CPU内部の状態を表すフラグの1つとして**割り込みイネーブルフラグ**があります．このフラグは割り込みが許可されているかどうかを示すフラグで，セットされていれば割り込みは許可されていることになります．リセットされていると割り込みは許可されない，つまり禁止されていることを示します．この状態では周辺機器から割り込みが発生しても，CPUはそれを受け付けません*．

割り込みイネーブルフラグを操作するために，専用のマシン語命令が用意されています．割り込みイネーブルフラグをセットする命令がSTI(SeT Interrupt enable flag)命令で，リセットする命令がCLI(CLear Interrupt enable flag)命令です．言い換えると，STI命令は割り込みを許可する命令で，CLI命令は割り込みを禁止する命令ということになります．

*ただし，NMI(Non Maskable Interrupt：マスクできない割り込み)という特殊な割り込みだけは禁止できない．

3

アセンブラ・プログラミングの基礎

アセンブラはマシン語プログラムを作成するためのツールですが，マシン語を知っているだけではプログラムを作成することはできません．アセンブラの世界はマシン語の世界そのものではなく，マシン語を扱いやすいように概念的に表現した世界だからです．

MASM は，プログラミングを効率よく行うための便利な機能をたくさん備えています．本章ではそのなかから，COM モデルのプログラムを作成するために最低限必要となる知識について解説します．本章で解説することは，アセンブラとしての一般的な知識であり，MASM 以外のアセンブラでもほぼ共通のものです．

3.1

アセンブラとマシン語

「アセンブラ」は，アセンブリ言語で書かれたプログラムをマシン語へと変換してくれるプログラミングツールです．しかし，それだけではアセンブラの役割をわかったことにはなりません．本節ではアセンブラとマシン語の関係を明らかにすることによって，アセンブラの役割を解説していきます．

マシン語命令と擬似命令

マシン語のプログラムは，図 3-1 のようにマシン語の命令 1 つ 1 つに 1 対 1 で対応しているアセンブリ言語のニーモニックで表します．これを CPU が読み込んで実行するマシン語に変換するのが「アセンブル」という操作です．そして，このアセンブルという作業を行ってくれるのが**アセンブラ**にほかなりません．

図 3-1 の左側のニーモニックで表されたプログラムを「ソースプログラム」と呼び，右側のアセンブルしてできるマシン語プログラムを「オブジェクトプログラム」と呼びます．

ある種の低級なアセンブラ，たとえば SYMDEB(または DEBUG)の A コマンドでも，アセンブリ言語のニーモニックで書いたソースプログラムを右

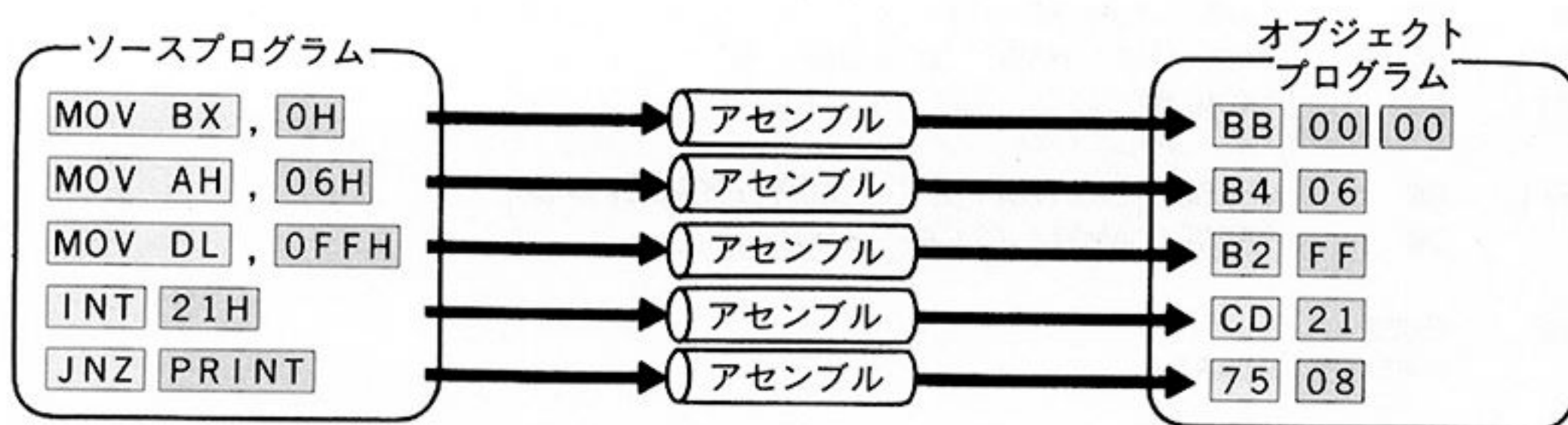



図 3-1　アセンブルとは

のようなマシン語に変換することができます．ところが，現実のプログラム開発では，このような低機能のアセンブラを使うことは，まずありません．実際にはもっと高級なアセンブラが使用されます．

高級なアセンブラでも，マシン語に直接対応するニーモニックをマシン語に変換する，つまりアセンブルすることが仕事の中心です．しかし，それ以外にも実にさまざまな仕事を引き受けてくれます．

アセンブル以外の仕事をアセンブラにお願いするには，ソースプログラムにニーモニック以外の命令を書かなければなりません．たとえば，1章で紹介したソースプログラムは図3-2のようなものでした．

```
                                                        枠
        ASSUME  CS:CODE,DS:CODE
CODE    SEGMENT

        ORG     100H
START:  MOV     BX,0
NOINPUT:MOV     AH,06H                           中身
        MOV     DL,0FFH
        INT     21H
        JNZ     PRINT
        INC     BX
        CMP     BX,5
        JGE     START
        JMP     NOINPUT
PRINT:  SHL     BX,1
        MOV     DX,TABLE[BX]
        MOV     AH,09H
        INT     21H
        MOV     AH,4CH
        MOV     AL,00H
        INT     21H

HARE    DB      'HARE',0DH,0AH,'$'
KUMORI  DB      'KUMORI',0DH,0AH,'$'
AME     DB      'AME',0DH,0AH,'$'
ANOTI   DB      'AME NOTI HARE',0DH,0AH,'$'
KNOTI   DB      'KUMORI NOTI AME',0DH,0AH,'$'

TABLE   DW      OFFSET HARE,OFFSET KUMORI,OFFSET AME
        DW      OFFSET ANOTI,OFFSET KNOTI

CODE    ENDS
        END     START
```

図3-2 プログラムの中身と枠

図3-2のソースプログラムのうち，直接マシン語と対応する部分，つまりニーモニックの部分をプログラムの「中身」，それ以外の部分を「枠」と呼びましょう．プログラムの中身は図3-1のようにアセンブルされ，オブジェクトプログラムに変換されます．ところがプログラムの枠はアセンブルするとどこかへ消えてなくなってしまいオブジェクトプログラムには形を残しません．

プログラムの中身と枠は，実は次の**図3-3**に示すような関係になっています．プログラムの中身はCPUへの命令であり，オブジェクトプログラムがCPUに読み込まれることによって初めて実行されます．図に示すように，プログラムの中身は機械的にオブジェクトプログラムへ変換されるだけです．これに対し枠の部分は，CPUへの命令ではなくアセンブラに対する命令であり，アセンブルを行うときにその命令が実行されます．

アセンブリ言語のソースプログラムは，このようにCPUへの命令とアセンブラへの命令が入り混じったものになっています．プログラムの枠の部分であるアセンブラへの命令は，プログラムの本質であるCPUへの命令と区別するために**擬似命令**と呼びます．

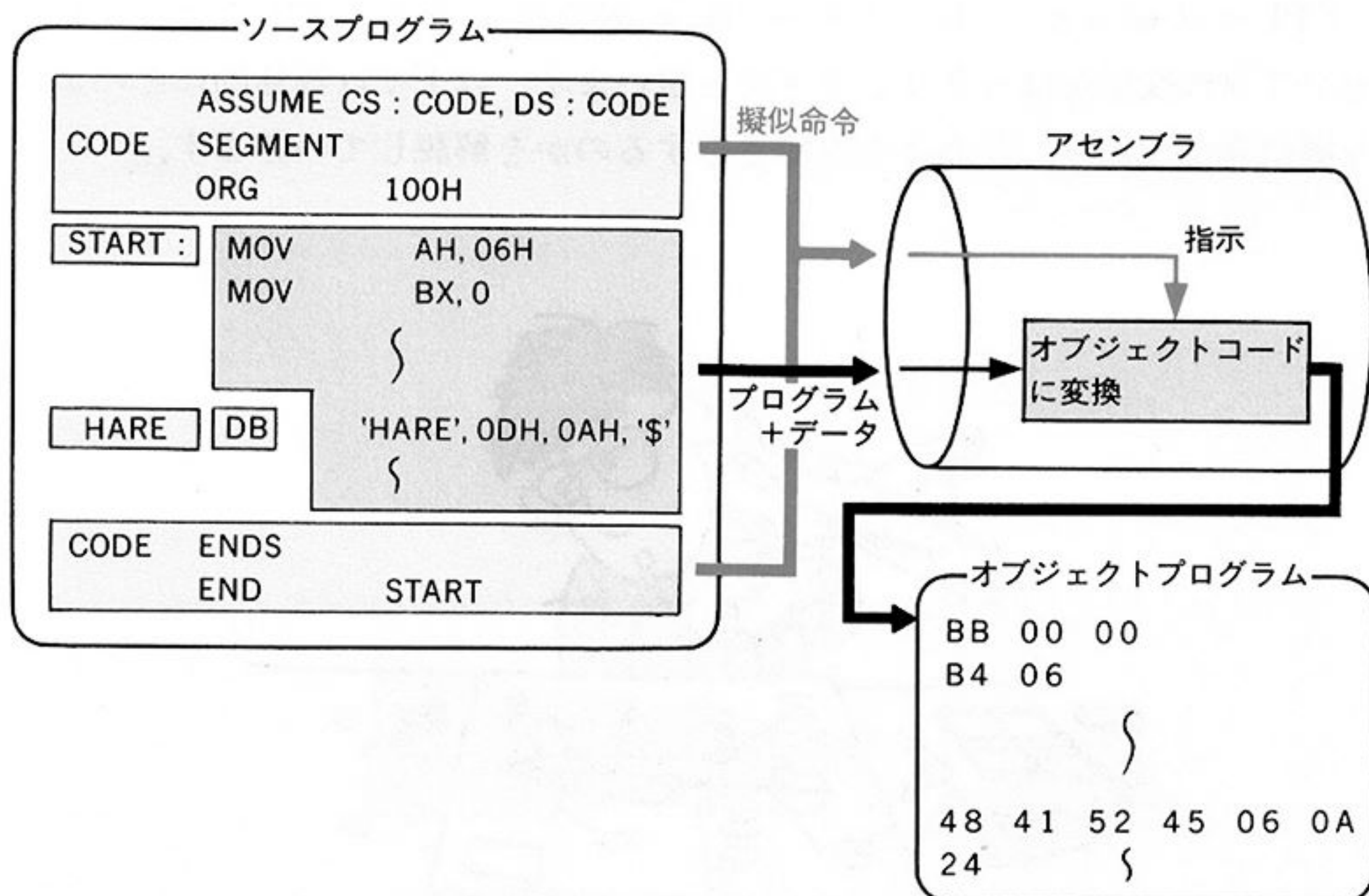



図3-3 CPUへの命令とアセンブラへの命令（擬似命令）

擬似命令の役割

擬似命令とはいったい何を命令するのでしょうか．CPUへの命令だけでなく，擬似命令までも使うとかえって面倒になるように思われますが，そうではありません．擬似命令を使うことによって便利で効率のよいプログラム開発が可能になります．

マシン語は数値の羅列にすぎません．これを人間にわかりやすいように意味のある記号に置き換えたのがアセンブリ言語のニーモニックです．同様にメモリやアドレスなどCPUに関連するさまざまな概念をわかりやすく表現するのが擬似命令の役割の１つです．

また，マシン語の命令は１つ１つが非常に簡単な機能しか持たず，いくつも組み合わせなければ目的の動作を達成できません．簡単な動作を記述するにも面倒になりがちで，プログラムをざっと眺めてもどういう動作をするのかよくわからない場合も少なくありません．このようなアセンブリ言語の欠点を補い，プログラムを効率よく，しかも読みやすく記述する助けをするのも擬似命令の役割の１つです．

CPUへの命令と，アセンブラへの命令（擬似命令）を区別することによりアセンブラの役割がはっきりしてきたと思います．以下では具体的にどのような擬似命令があり，どのような働きをするのかを解説していきます．

3.2 数値表現とラベル

アセンブラでプログラムを作成するために知っておかなければならない最低限の擬似命令について以降の節で解説します．ここで解説する擬似命令を理解すれば，MS-DOSの実行可能ファイルのうちの「COMモデル」と呼ばれる形式のプログラムを作成することができるようになります．

COMモデルのプログラムを作成するためには，図3-4に挙げた擬似命令が必要です．これだけの擬似命令をマスターすれば，たいていのプログラムなら書くことができます．むつかしい命令は1つもありません．1つ1つじっくりと解説していきますから，それぞれの役割をしっかり把握してください．

図3-4 COMモデルのプログラムを作成するために必要な擬似命令

数値表現

擬似命令ではありませんが，アセンブラでプログラムを作成するための文法上の約束事として覚えておかなくてはならないのが，数値の表現方法です．

MASMでは，数値は10進数で表します．DEBUGやSYMDEBでは，数値は何も指定しなければ16進数となりますが，MASMでは10進数として扱われるので注意してください．16進数は数値の末尾に「H」を付けます．たとえば，

```
00H, 10H, 41H, 5FH, 0C0H, 0FFH
MOV  AH, 4CH
```

のようになります．

ここで，先頭の桁が数字でなくA～Fである数値には先頭に「0」を付けなければなりません．これは，MASMではラベル（次項で解説）と区別するために，数値は必ず数字で始まると仮定されているからです．

文字コードを数値として扱いたい場合は，文字を「'」（シングルクォーテーションマーク）または「"」（ダブルクォーテーションマーク）で囲みます．たとえば，

```
'A', 'B', 'C', '1', '2', '(', ' " ', " ' ", 'ア', 'イ'
MOV  AL, 'A'
```

のようになります．この例でわかるように，クォーテーションマーク自身の文字コードを使いたい場合，もう1種類のクォーテーションマークで囲みます．

ラベル

［書式］ 定義　ラベル名：

参照　JMP　ラベル名

・ラベル名は＜アルファベット，@，$，_，?，数字＞からなる文字列で，数字で始まることはできない．

ラベルは擬似命令のなかでも最も理解しやすいものでしょう．ラベルはメモリのアドレスをわかりやすく表現したものです．

図3-5のように，アセンブリ言語ではプログラム中で使用するアドレスに名前を付けて，その名前でアドレスを表すことができます．そして，その名前のことをラベルと呼びます．ラベルを定義するには，名前を付けたいアドレスのところ，つまり名前を付けたい場所にある命令の前で，

```
名前:
```

とします．名前の後に「：」(コロン)を付けるとその名前のラベルを定義したことになります．そして，定義の次に出てくるCPUへの命令のアドレスがそのラベルに対応するアドレスです．定義したラベルはジャンプ命令などで飛び先として使用することができます．たとえば1章のプログラムでは，

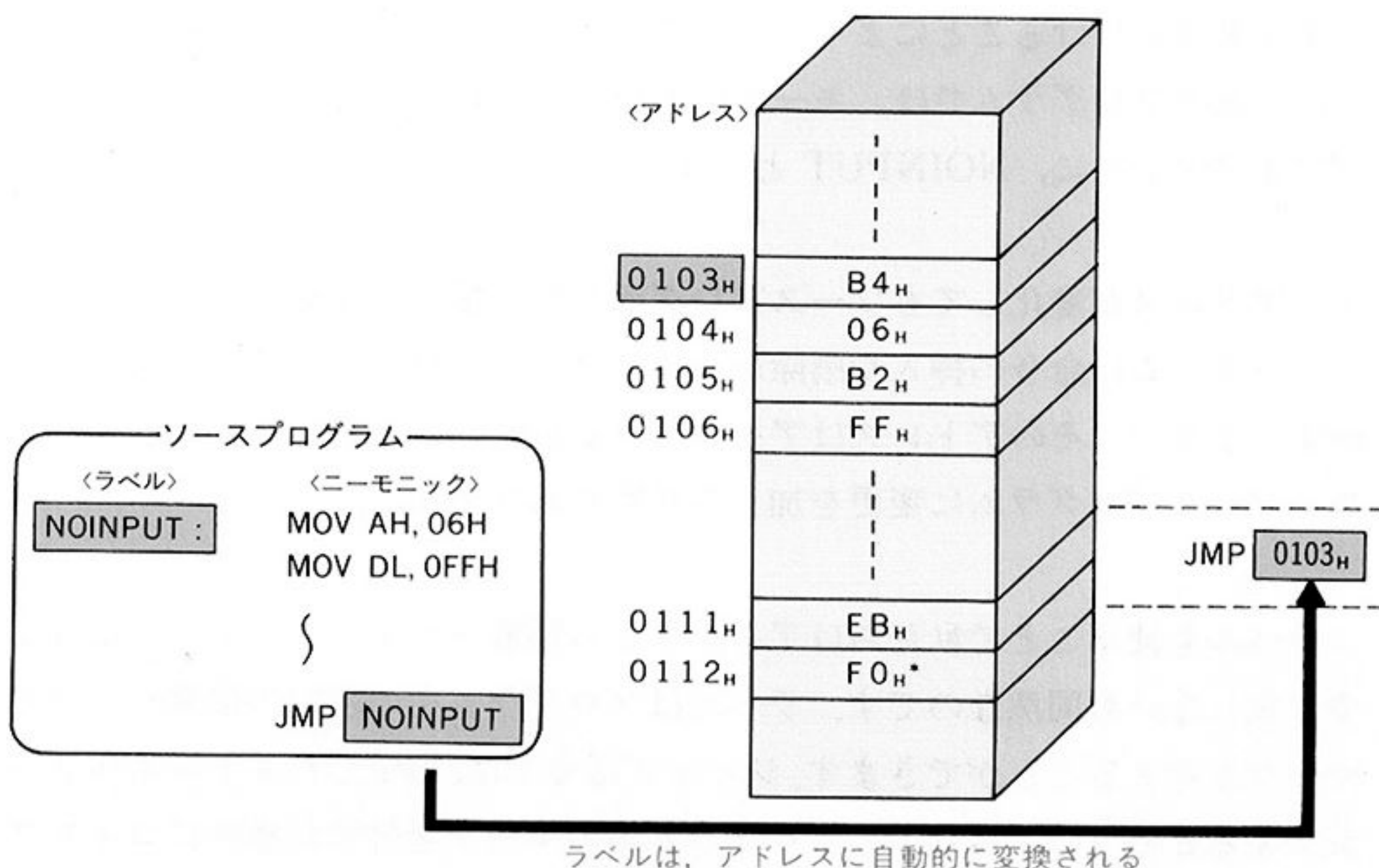
```
NOINPUT:
        ⋮
     JMP  NOINPUT        ……①
```

＊ジャンプ先のアドレスは相対アドレスに変換される(前書「はじめて読む8086」を参照)

図3-5　ラベルとアドレス

としていますが，①のニーモニックに対して，NOINPUT という名前のラベルに対応するアドレスへの無条件ジャンプ命令がマシン語として出力されます．

ラベルを使用することにより，次のような大きなメリットが生まれます．

1．アドレスを計算しなくてもよい

アセンブラはアセンブルを行いながら常に変換したマシン語のバイト数を数え，次の命令のアドレスを算出しています．ですからラベルの定義があればその値とラベルの名前を対応づけることができるのです．

もしもラベルを使わないとしたら，私たちはアセンブル後のマシン語のバイト数を数えてアドレスを算出しなければなりません．

2．アドレスに意味のある名前を付けることができる

ラベルには好きな名前を付けることができます．そのアドレスにはどういう時にジャンプしてくるかによって，その条件を表す名前を付けることができるのです．たとえば条件ジャンプ命令でジャンプする飛び先にはその条件を表す名前を付けることによりソースプログラムはよりわかりやすくなります．1章のプログラムでは，キーボードからの入力がなかった場合にジャンプするアドレスに，NOINPUT というラベルを付けています．

3．アドレスが変化してもソースプログラムに変更がいらない

プログラムに命令の挿入や削除があるとラベルに対応するアドレスに変化が生じますが，そのアドレスはアセンブラが自動的に計算してくれるものなのでソースプログラムに変更を加える必要がありません．

ラベルを使うことで私たちはアドレスから解放されます．アドレスはもはや存在しないも同然なのです．ラベルはプログラム中の特定の位置につけたマークと考えることができます．ジャンプ命令では，指定したアドレスへジャンプするというのではなく，プログラム中のマークを付けた場所にジャンプするという考え方でプログラムを作成することができます．

3.3

ORG 擬似命令と END 擬似命令

ORG 擬似命令

［書式］ ORG　数値

ORG 擬似命令は ORiGin（起源）の略であり，プログラムの始まるアドレスを指定するために使います．ラベルに対応するアドレスを算出するために，アセンブラは常に次のマシン語命令のアドレスを算出していることはすでに解説しました．ORG 擬似命令はこの値を強制的に指定した値に変更してしまうのです．

その結果，ORG 擬似命令の次に出てくるマシン語命令は，指定した値のアドレスに置かれます．そして，以降のマシン語命令は続くアドレスに置かれていきます．

プログラムの先頭で，つまりマシン語命令が1つも登場しないうちに ORG 擬似命令が使われた場合，プログラム全体が指定したアドレスから始まるメモリに置かれることになります．このように，ORG 擬似命令はプログラムの先頭で使用し，プログラムの先頭アドレスを指定するための擬似命令だと思ってかまいません．

この命令は MS-DOS の規約を守るために必要となります．COM モデルのプログラムは，必ずアドレス 0100_H から始まらなければなりません（くわしくは 4.8 節で解説）．ソースプログラムをアセンブルする際にも，先頭の命令が 0100_H からのメモリに置かれるようにアセンブルする必要があります．

したがって COM モデルのプログラムを作成するには，ソースプログラム中ですべてのマシン語命令に先立って，

```
ORG 0100H
```

と指定します.

アドレス0100_Hより前の100_H(256)バイトはPSP(Program Segment Prefix)という領域です.この領域はMS-DOSがプログラムを管理したり,プログラムに情報を渡すために使用します.4章でも解説しますが,COMモデルのプログラムはこのPSPに続くアドレス0100_H以降のメモリにロードされ,0100_Hから実行が開始されるので,ORG擬似命令を指定しなければならないのです.

ORG擬似命令を指定しなければどうなるのでしょうか.何も指定がなければ,アセンブラはアドレス0からプログラムが始まるものとしてアセンブルを行います.つまり,**図3-6**のように最初の命令はアドレス0に置かれ,以下のアドレスに次の命令が続くと仮定します.

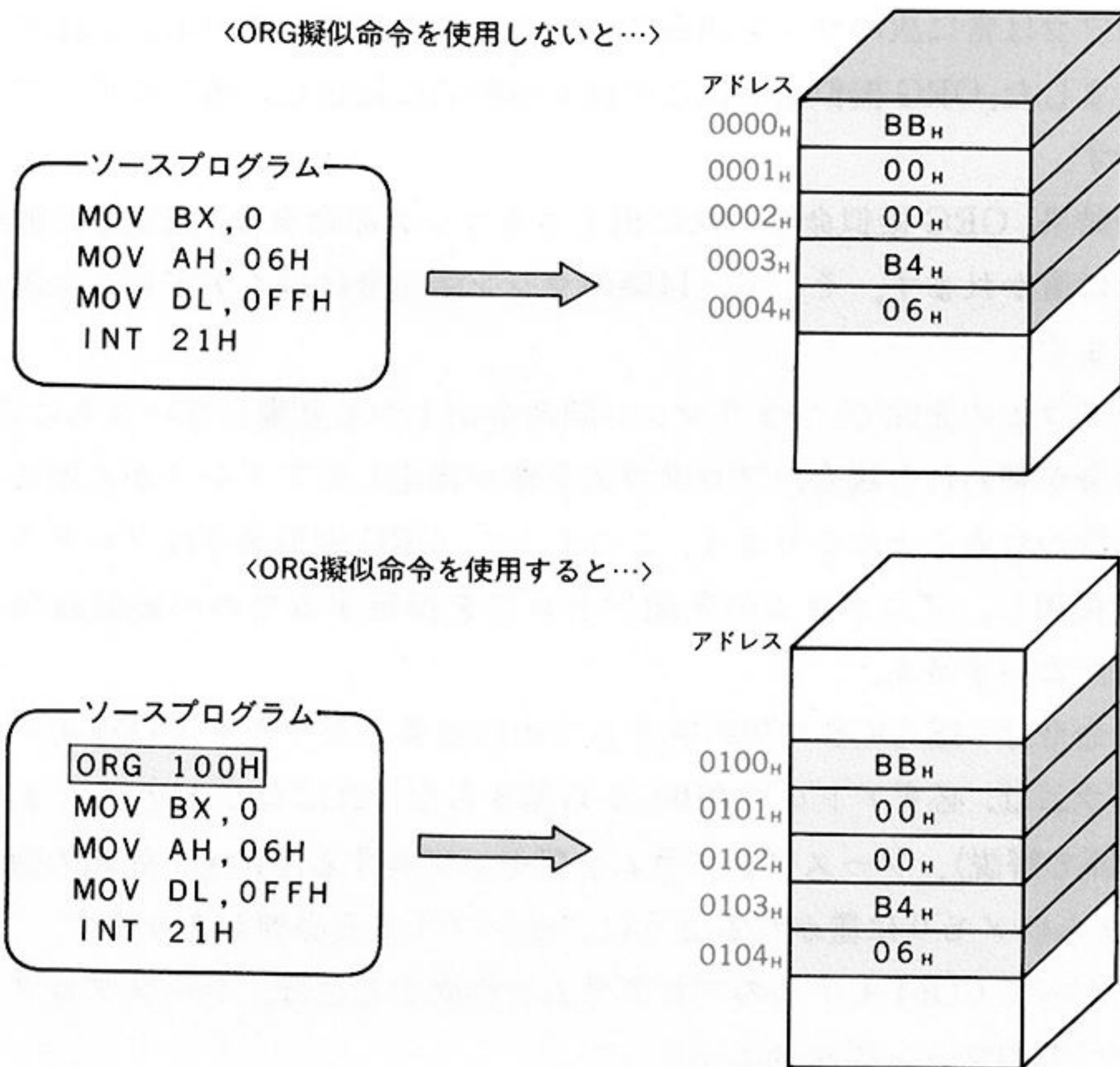



図3-6　ORG擬似命令で開始アドレスを指定する

ただし，ORG擬似命令を指定しない場合のアドレスは，あくまで「仮の」アドレスにすぎません．0から始まるように固定されるわけではなく，とりあえず0から始まるアドレスにしておくだけです．くわしくは5章で解説しますが，リンカが実際にロードされるアドレスに合わせて改めてアドレスを振り直します．このことはプログラムをいくつものモジュールに分けてアセンブルする，分割アセンブルにおいて非常に重要な意味を持ちます．

END擬似命令

［書式］ END ラベル
　　　　 END

映画のラストシーンには「END」や「FIN」という文字が大写しになりますが，アセンブラにもプログラムのおしまいを伝えてやらなければなりません．ソースプログラムの最後には必ずこのEND擬似命令を書きます．

END擬似命令では，プログラムのスタートアドレスをパラメータとして指定します．指定したラベルがそのプログラムの実行開始アドレスになります．ただし，COMモデルのプログラムでは実行開始アドレスを自由に選ぶことはできません．実行開始アドレスは0100_Hに固定されているからです．したがって，ORG擬似命令の直後にラベルを定義し，そのラベルをEND擬似命令で指定しなければなりません．

4章で解説するEXEモデルのプログラムでは，任意のラベルを実行開始アドレスとして指定することができます．また，5章で解説する分割アセンブルを行う場合には，プログラムの実行開始アドレスの含まれるメインモジュール以外では，ラベルを指定する必要はありません（次ページの図3-7を参照）．

<COMモデル>

```
    ASSUME CS : CODE
    ASSUME DS : CODE
CODE SEGMENT
        ORG 100H
START:
        ～
CODE ENDS
        END START
```

ORG 100Hの直後のラベルを指定する

<EXEモデル*>

```
    ASSUME CS : CODE
    ASSUME DS : CODE
CODE SEGMENT
        ～
START:
        ～
        END START
```

任意のラベルを指定できる

*EXEモデルについては4章で，分割アセンブルについては5章で解説する

図 3-7　END 擬似命令の役割

3.4

データ定義擬似命令

DB 擬似命令

［書式］ ラベル DB データ
DB データ

マシン語プログラムはCPUへの命令だけからなるわけではなく，一般にデータ部分が必要です．たとえば，画面にメッセージを出力するならメッセージの文字列がデータとして必要であり，入出力を行うなら一時的にそのデータを格納しておく領域が必要です．

1章のプログラムでは，次ページの図3-8のように命令コードとデータ部分がそれぞれ存在します．データ部分は表示するメッセージの文字列と，そのアドレスを並べた配列です．

図3-8に示すように，プログラムの命令コード部分はニーモニックで表します．そしてデータ部分は**データ定義擬似命令**を使って表します．DB擬似命令はデータ定義命令の1つで，1バイトデータを定義するための命令です．

DB擬似命令は，マシン語命令のニーモニックをマシン語コードに変換するのと同様に，指定された数値をそのままオブジェクトファイルに出力します．また，データを定義する以外にも，単にデータ領域を確保するための役割も持っています．データ部分はプログラムの一部であり最終的な実行ファイルにも含まれるので，指定したデータはプログラムの「中身」であるといえます．DB擬似命令は，**表3-1**に示すようにいろいろな形式で使うことができます．

DB擬似命令で指定するデータ形式を，それぞれをくわしく解説していきましょう．

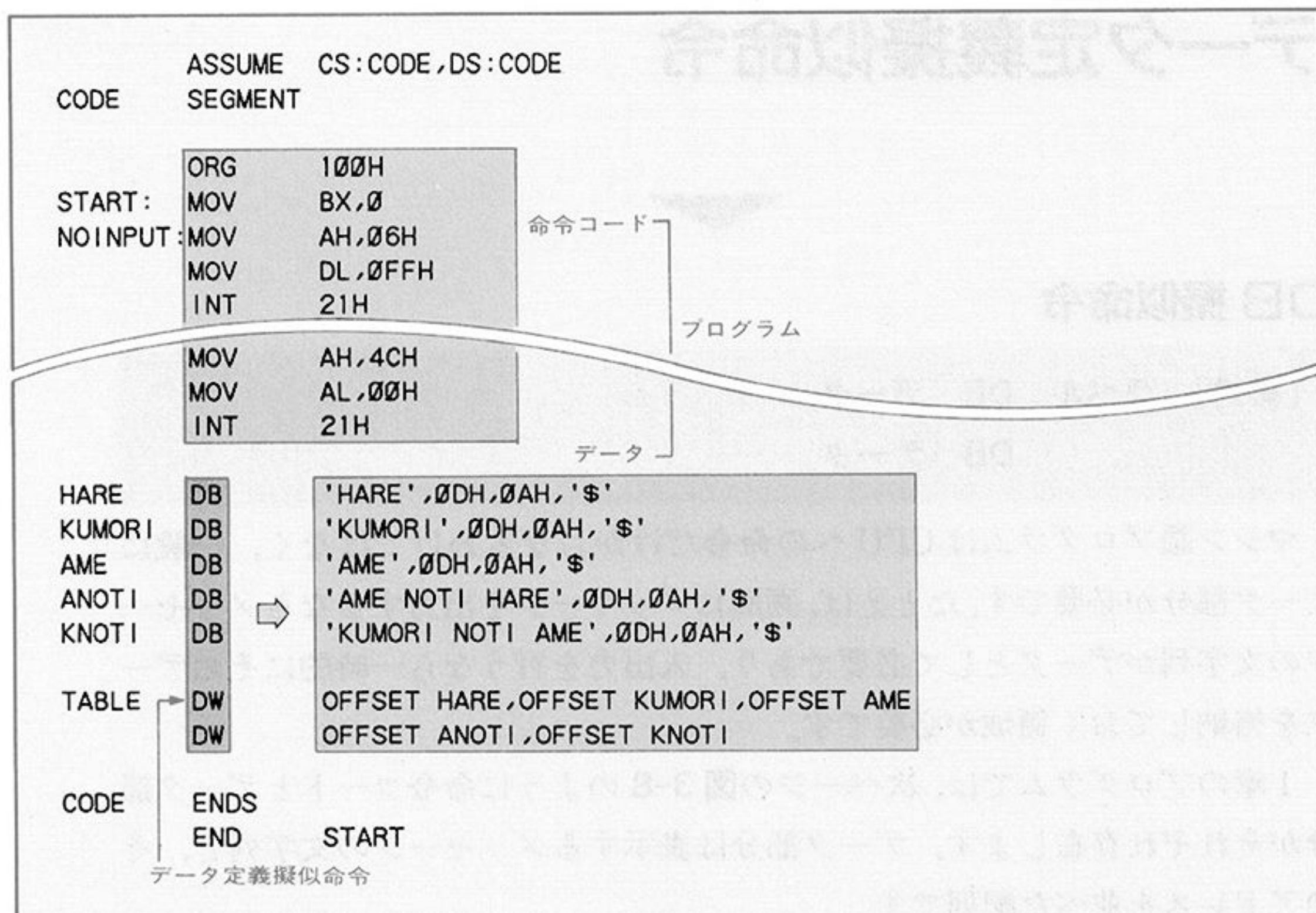



図 3-8　プログラム＝命令コード＋データ

DB擬似命令の書き方	データ領域の定義
DB 1	01H
DB 8, 0, 8, 6	08H 00H 08H 06H
DB 10 DUP(0)**	00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
DB 3 DUP(0, 1, 2)**	00H 01H 02H 00H 01H 02H 00H 01H 02H
DB 'A'	41H
DB 'Hajimete',0DH, 0AH,'$'	48H 61H 6AH 69H 6DH 65H 74H 65H 0DH 0AH 24H
DB ?	□ *
DB 5 DUP(?)**	□□□□□ *

＊何が入っているかは不定
＊＊カッコは必ず必要

表 3-1　DB擬似命令の例

DUP 擬似命令

［書式］ 繰り返し数 DUP （初期値）

同じ値をいくつも続けてデータとして置いておきたい場合には，同じ数字を並べて定義する方法もありますが，それを簡略化する方法があります．これには，表3-1の3，4番目の例のようにDUP擬似命令を併用します．DUPはDUPlicateの略であり，複製を作るという意味です．

文字コードの定義

数値だけでなく，文字や文字列もセットしておくことができます．たとえば，表3-1の5番目の例のように，

```
DB 'A'
```

と指定すると，Aという文字の文字コード41_Hがセットされます．結局，

```
DB 41H
```

と同じ結果になります．

文字列の定義

文字コードを複数並べて定義する場合には，数値を並べて定義するときのように「,」(カンマ)で区切って並べることもできますが，文字列として定義することもできます．次の例のように，クォーテーションマークの間に文字をいくつも並べてしまえばよいのです．

```
DB 'H', 'A', 'R', 'E'  →  DB 'HARE'
```

これを利用すれば，漢字を含む日本語の文字列も，

```
DB '明日は晴れです'
```

のようにデータとして定義することができます．

データ表現の混在

数値，文字，文字列の定義は，「，」（カンマ）で区切って混在させることもできます．たとえば，表3-1の6番目の例は文字列の後に改行コード（$0D_H$，$0A_H$）を定義し，さらに文字を1つ定義したものです．改行コードは表示できない文字なので，数値として定義します．

領域のみの確保

〝キーボードから入力された文字を一時的に格納しておくためのデータ領域を確保しておきたい〞というように，内容は指定しないが領域だけ確保しておきたい場合があります．この場合には，適当な値を指定してデータ領域を確保してもかまいませんが，初期値はなんでもよく領域を確保することが必要であることをはっきり示すために，数値の代わりに「？」（クエスチョンマーク）を書きます．数バイトの領域をいっぺんに確保しておきたいという場合には，表3-1の8番目の例のようにDUP擬似命令を使います．

DW擬似命令

DB擬似命令は1バイトの領域を確保する命令でしたが，1ワード，つまり2バイトの領域をいっぺんに確保するDW擬似命令も用意されています．1ワードのデータ領域は，1バイトでは表しきれない大きな数値や，アドレスの値を格納するために必要です．

DW擬似命令はDB擬似命令とほとんど同じ役割を持っており，書き方もまったく同じです．違うのは確保される領域の大きさだけです．その違いを示したのが，**表3-2**です．

DW擬似命令の例	データ領域の定義
DW 6, 8, 10H, 36ADH	06H 00H 08H 00H 10H 00H ADH 36H
DW 5 DUP (?)	□□□□□□□□□□ *

＊何が入っているか不定
〈注意〉：80系CPUの特徴として下位バイト，上位バイトの順に並ぶ

表3-2　DW擬似命令の例

DB擬似命令とまったく同じ数値を並べても，1つの数値に対して2バイト分の領域が確保されます．80系のCPUでは1ワードのデータは，表のように上位バイトがアドレスの高い方，下位バイトが低い方に格納されます．したがってワードデータがいくつも並ぶと，下位バイト・上位バイト，下位バイト・上位バイト，と並ぶことになります．

その他のデータ定義命令

DB，DW擬似命令以外にも，データを定義するための擬似命令があります．これらは主に特定のデータ型のデータを格納する領域を確保する役割をもっています．これらの擬似命令を**表3-3**に示しました．

データ定義擬似命令	意　味
DB(Define Byte)	1バイト(8ビット)のデータ領域を割り当てる
DW(Define Word)	1ワード(2バイト)のデータ領域を割り当てる
DD(Define Double word)	2ワード(4バイト)のデータ領域を割り当てる
DQ(Define Quad word)	4ワード(8バイト)のデータ領域を割り当てる
DT(Define Ten byte)	5ワード(10バイト) のデータ領域を割り当てる

表3-3　データ定義擬似命令

バイト型のデータはメモリの最小単位であり，必要なバイト数だけ自由に定義することができるので，いわばオールマイティなデータ型です．ワード(2バイト)型のデータ定義は主にバイト型では表せない範囲の数値や，アドレスを格納するために使われます．ダブルワード(4バイト)型のデータ定義は，32ビット整数やセグメントアドレスを含めたアドレス(4章で解説)を格納するために使われます．

それ以上のバイト数を定義する擬似命令は，実数(小数点以下の桁を持つ数)を定義するために使われます．マシン語で実数を扱う方法やデータ表現の方法については本書では解説しません．

3.5 データラベル

データラベルの定義

［書式］ 定義 データラベル名 DB データ
参照 MOV AL, データラベル名

・データラベル名は<アルファベット, @, $, _, ?, 数字>からなる文字列で，数字で始まることはできない．

データ部分にも，命令部分と同じように，ラベルを付けることができます．もう一度ラベルの意味を思い出すために，ラベルの役割を図解しておきましょう（図3-9）．データ部分に定義するラベルもやはりアドレスに名前を対応させて表す手段です．

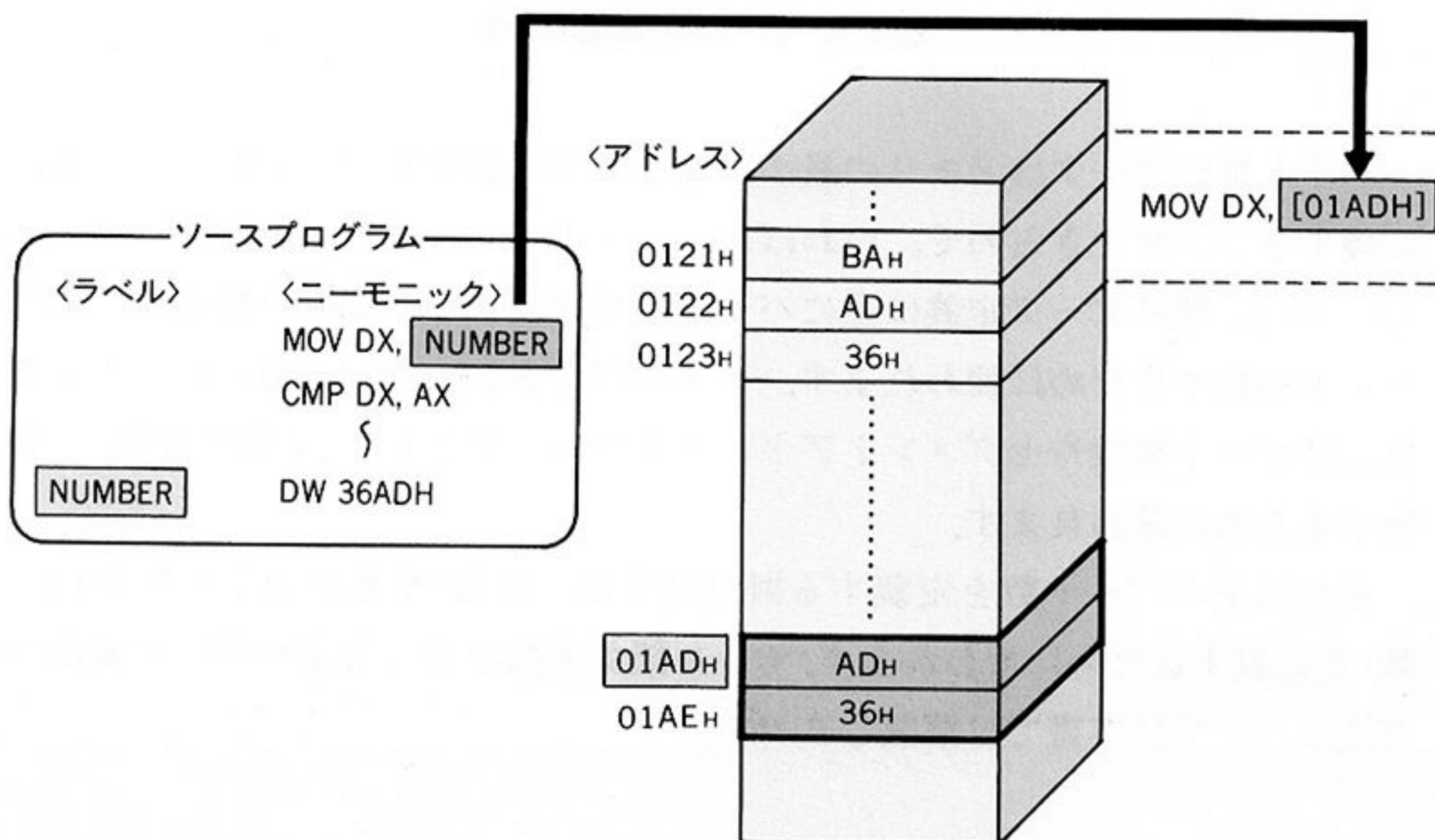



図3-9 データラベルの役割

データを定義する擬似命令と同じ行の先頭に名前を付けることで，そのデータに名前を付けたことになります．本書ではデータに付けるラベルのことを**データラベル***と呼ぶことにします．区別をはっきりさせるために，命令に付けるラベルは**コードラベル**と呼ぶことにします．

データラベルは，コードラベルと同じように定義しますが，注意しなければならないのは，データラベルを定義する時には「：」(コロン)を付けないことです．しかも，必ずDBなどのデータを定義する擬似命令と同じ行に書かなければなりません．コードラベルは，必ずしも命令と同じ行に定義する必要はありません．

データラベルの参照

データラベルは，マシン語命令のニーモニックのなかで，データを格納するメモリを参照するために使用することができます．次の図3-10を見てください．この図では，違いを対比させるために次節で解説するOFFSET擬似命令の役割を併せて図解してあります．

この場合，MESSAGEというデータラベルは$01C3_H$というアドレスに対応しています．コードラベルの場合は，ラベル名がアドレスそのものと対応していましたが，データラベルの場合にはちょっと違います．このことは重要ですからよく覚えておいてください．

たとえば，SYMDEBのようにアドレスを直接数値で指定する場合，そのアドレスのメモリの内容を参照するには，

```
MOV  DX, [01C3H]
```

のようにアドレスを [ ] で囲みます．これに対し，アセンブラのソースプログラムでは，

```
MOV  DX, MESSAGE
```

*「データラベル」は本書だけの用語で一般的なものではない．MASMのマニュアルなどでは「変数」と呼んでいる．次節で示すように，データラベルが命令中で使われると，アドレスではなくそのアドレスのメモリの内容を示すため，高級言語における変数と同じような書き方ができるからである．本書では，5章で示すEQU擬似命令による定数定義との混乱を避けるために，ラベルという言葉で統一して扱っている．

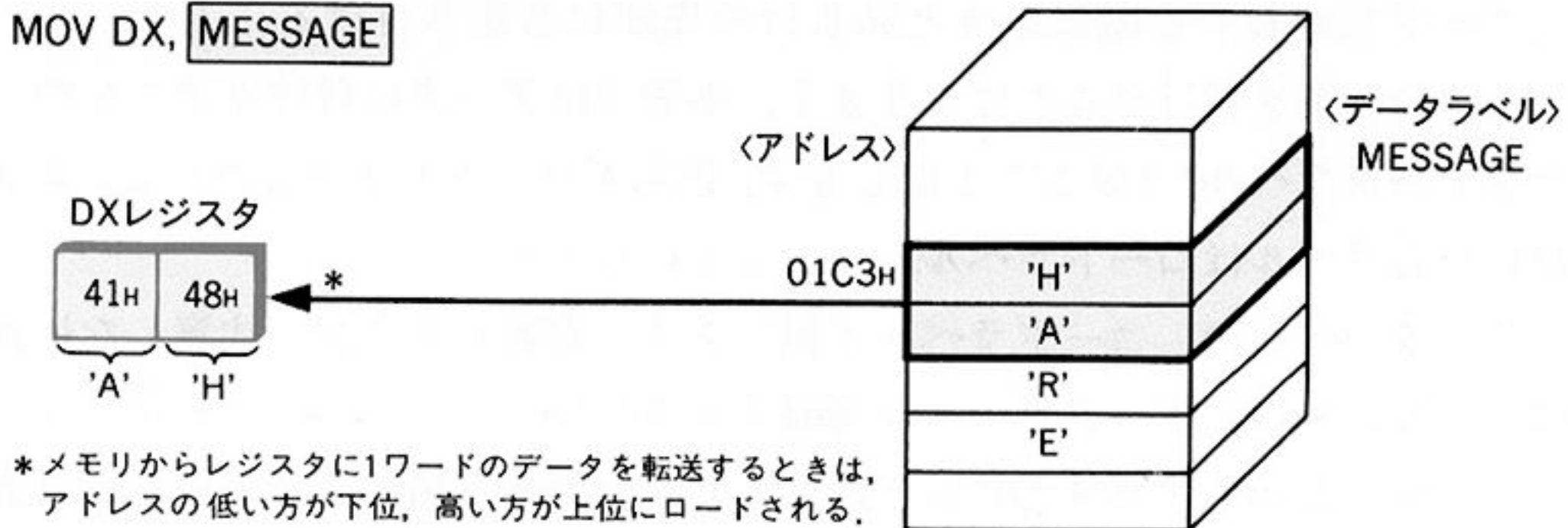
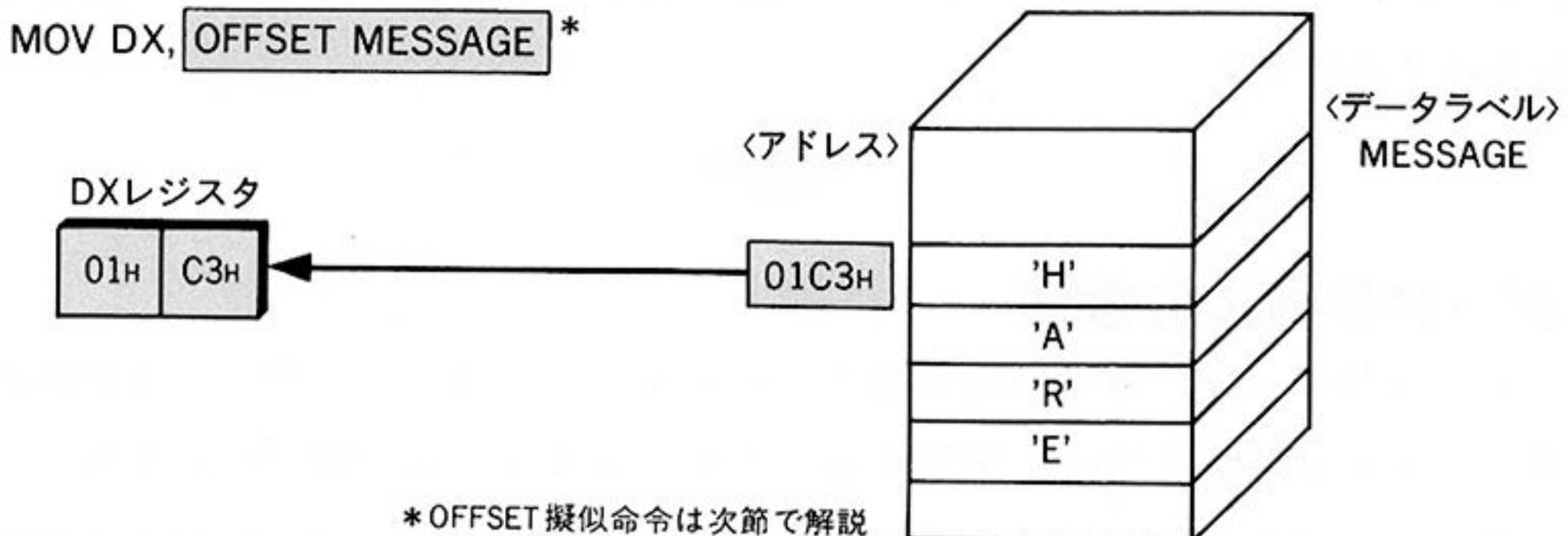



図 3-10　データラベルの参照

とします．このことからわかるように，データラベルはラベルに対応するアドレスそのものではなく，そのアドレスが示すメモリの内容を表します．つまり，データラベルそのものを指定すると，アドレスの値を転送したり演算したりする命令ではなくて，メモリに対して転送や演算を行う命令になります．もう一度図 3-10 を見てじっくり確認してください．

配列の参照

データ定義擬似命令では，「,」(カンマ)で区切って複数のデータを並べて定義することができます．このようなデータの列はデータラベルを使って配列として参照することができます．たとえば，

```
CPU  DB  80, 65, 68
```

のような定義があるとすれば，3つのデータをそれぞれ，

CPU[0] …… 「(データラベルCPUのアドレス)+0」のアドレスのメモリの内容

CPU[1] …… 「(データラベルCPUのアドレス)+1」のアドレスのメモリの内容

CPU[2] …… 「(データラベルCPUのアドレス)+2」のアドレスのメモリの内容

として参照することができます．つまり，

```
MOV  AL, CPU [1]
```

という命令では，65というデータがALレジスタに転送されます．この場合，CPUというデータラベルに対応するアドレスが0200_Hであれば，

```
MOV  AL, [0201H]
```

という命令に変換されるわけです．

レジスタを使った配列の参照

SI, DI, BX, BPレジスタはポインタとして使用することができます*．すなわち，

```
MOV  AL, [BX]
```

のような使い方が可能です．これは，BXレジスタの内容をアドレスとするメモリの内容をALレジスタに転送するという命令です．このアドレッシングモードを使うと次のように記述でき，右のようにアセンブルされます．

```
MOV AL, CPU[BX]  →  MOV AL, [BX+0200H]
```

この場合，BXレジスタの値が0であればCPU [0]，BXレジスタの値が1であればCPU [1] を参照しているのと同じことになります．

*前書「はじめて読む8086」(アスキー出版局発行)を参照のこと

3.6 OFFSET 演算子と PTR 演算子

OFFSET 演算子

［書式］ OFFSET　ラベル

データラベルをニーモニック中で使用すると，データ部分のアドレスではなく，データを格納したメモリの内容を表しますが，そのメモリのアドレスはどうやって表すのでしょうか．図 3-10 で図解してあるので解説するまでもありませんが，データラベルに対応するアドレスは，次のように表します．

OFFSET　データラベル

OFFSET 演算子は，ラベルに対して使用する演算子です．ラベルの前にこの演算子を付けると，ラベルに対応するアドレスを表す値となります．

例題のプログラムでは，図 3-11 の部分で使われています．これは画面に出力するメッセージが格納されているアドレス値を，データとして定義している例です．

```
HARE    DB      'HARE',ØDH,ØAH,'$'
KUMORI  DB      'KUMORI',ØDH,ØAH,'$'
AME     DB      'AME',ØDH,ØAH,'$'
ANOTI   DB      'AME NOTI HARE',ØDH,ØAH,'$'
KNOTI   DB      'KUMORI NOTI AME',ØDH,ØAH,'$'

TABLE   DW      OFFSET HARE,OFFSET KUMORI,OFFSET AME
        DW      OFFSET ANOTI,OFFSET KNOTI

CODE    ENDS
        END     START
```

データラベルHAREのアドレスをデータとして格納することを表す

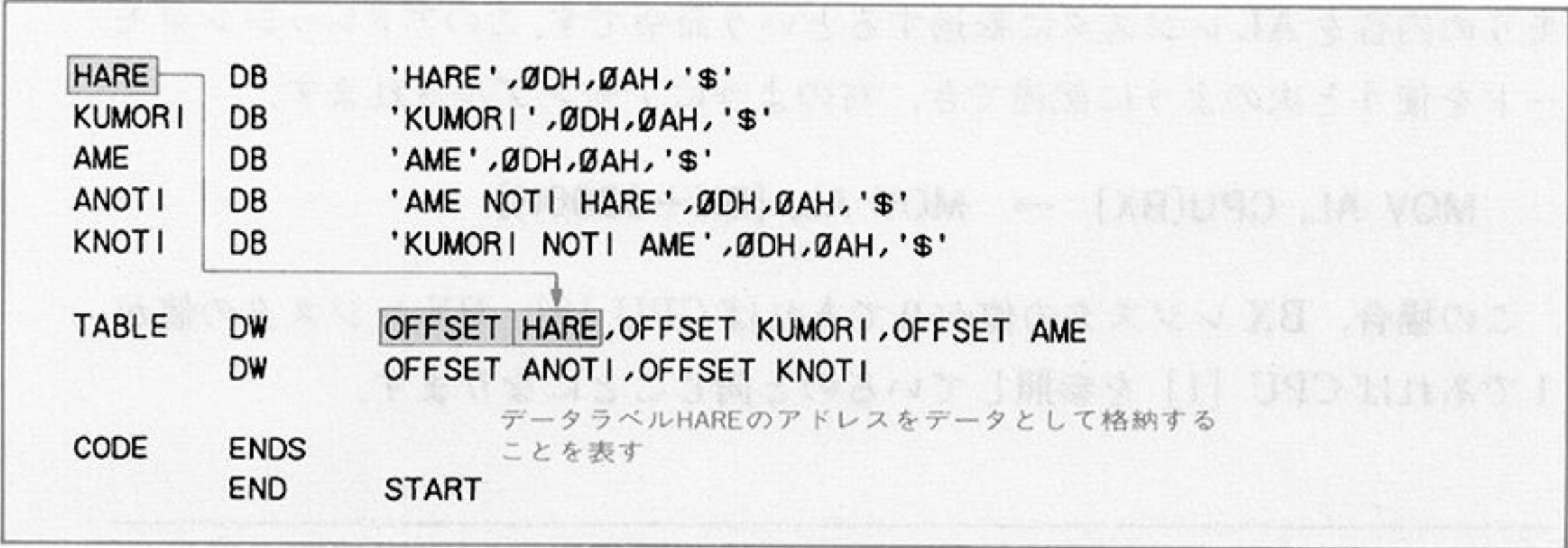



図 3-11　OFFSET 演算子の使用例

コードラベルとデータラベルについての解説が一通り終わったので，両者の違いを**表3-4**としてまとめておきます．

	コードラベル	データラベル
定義の例	PROG1: PROG2: MOV AX, 1	COUNT DW 0 STR DB 'HARE', 0DH, 0AH, '$'
参照方法	JMP PROG1 JGE PROG2	MOV CX, COUNT MOV DX, OFFSET STR
型	NEAR*	BYTE WORD DWORD など

*LABEL擬似命令を利用すれば，FAR型のコードラベルも定義できる．

表3-4　コードラベルとデータラベル

PTR演算子

［書式］ BYTE PTR　メモリアドレッシングまたは値
　　　　WORD PTR　メモリアドレッシングまたは値

MASMはデータの型を判別することができます．たとえば，データ転送命令などでAXレジスタが指定されれば，対応するレジスタやメモリ，データは2バイト，すなわちワード型でなければなりません．

```
MOV AX, BL
```

という命令は，AXがワード型でBLがバイト型なのでエラーになります．

レジスタとメモリ，またはデータというアドレッシングモードでは，使用するレジスタによって自動的にメモリやデータの型がバイト型であるかワード型であるかが判別されます．

```
MOV AX, 20H
```

という命令では，「20_H」は「0020_H」というワード型のデータとしてアセンブルされます．

このようにレジスタが使われる命令では，データの型をレジスタの種類から判別することができますが，「メモリ←データ」というアドレッシングモードなどでは，データの型がわかりません．

たとえば，**図 3-12** のような場合には，0 という値がバイト型なのかワード型なのかわかりません．つまり，この命令を SI レジスタの指している 1 バイトのメモリに 0 をストアする命令と解釈してよいのか，それとも SI レジスタの指しているメモリと続くメモリからなる 1 ワードのメモリに 0 をストアする命令と解釈してよいのかがわからないのです．

このような場合，MASM はデータの型を判別できないのでエラーとなります．これを解決するためにはデータの型をあらかじめ指定してやらなければなりません．このために用意されているのが PTR 演算子です．

PTR 演算子は図 3-12 のように，転送の対象となるメモリやデータに BYTE PTR(バイト型)や WORD PTR(ワード型)を付けることで型を指定します．この例のようにオペランドが 2 つある場合はどちらにつけてもかまいません．

このように，データの型が判別できないような場合には，必ず PTR 演算子でデータの型を指定します．次のような場合にもデータの型が判別できないので注意が必要です．

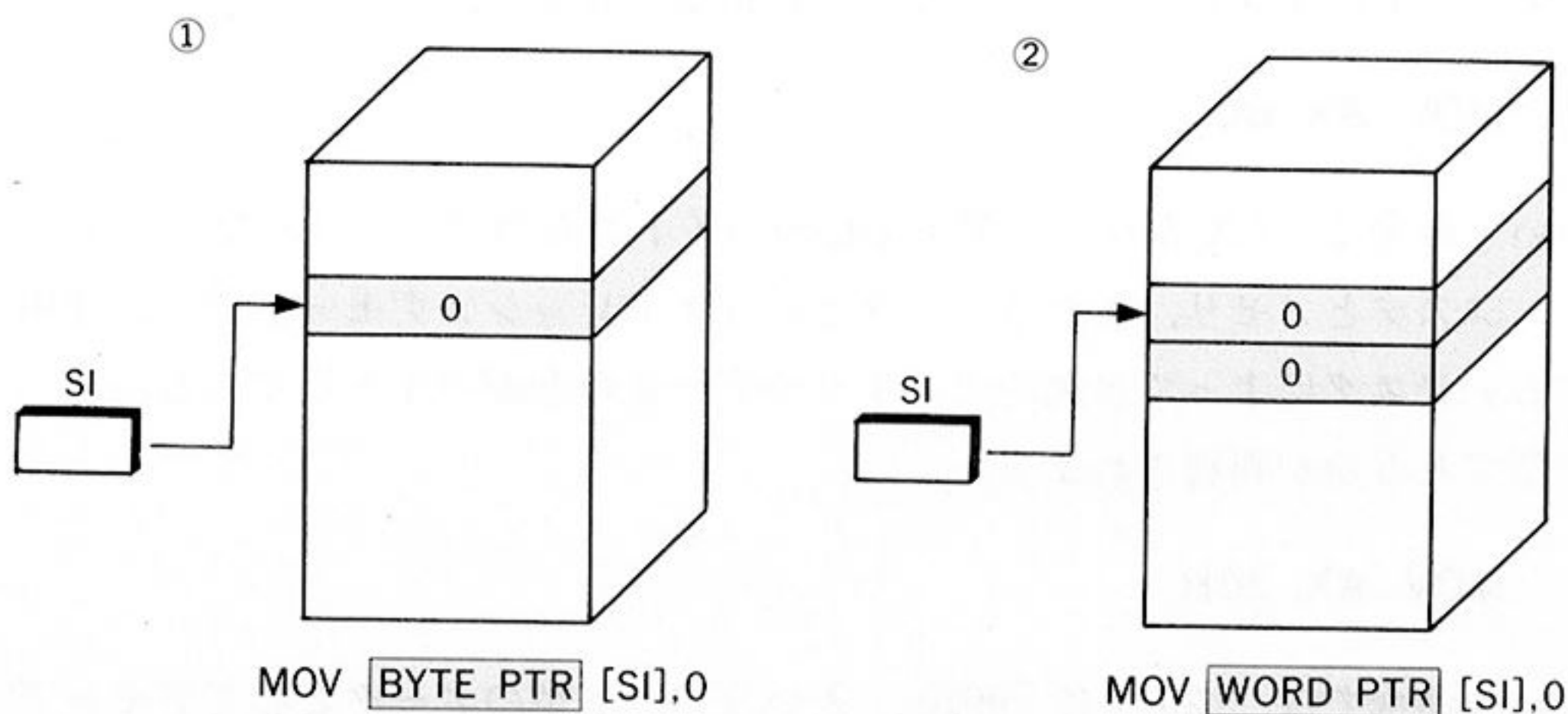



図 3-12　PTR 演算子の役割

```
CMP   [BX],1AH       → CMP   [BX],BYTE PTR 1AH
SHR   [SI],1         → SHR  BYTE  PTR  [SI],1
INC   [BX]           → INC  WORD  PTR  [BX]
```

データラベルの前方参照

データラベルは，DB 擬似命令で定義したか，DW 擬似命令で定義したかによってMASMが自動的に型を判別します.「COUNT　DW　0」と定義されたデータラベルに対して,

```
MOV  COUNT, 0
```

という命令があると，アドレスCOUNTと続く1バイトのメモリからなる1ワードのメモリ(これはDW擬似命令によって確保される)に1ワードデータ0を転送する命令と解釈されます.

したがって，データラベルに関しては型を指定する必要はないのですが，図3-13のようにデータラベルを前方参照している場合には注意が必要です．この場合，データラベルを参照している命令をアセンブルする時点ではアセンブラにはデータラベルの型がわかりません．MSAMはデータラベルの型がわからない場合は，とりあえずワード型と仮定してアセンブルを行います．データレベルが定義されBYTE型であることがわかると，あらためてアセンブルを行います(144ページ参照)．BYTE型に対するマシン語命令は,

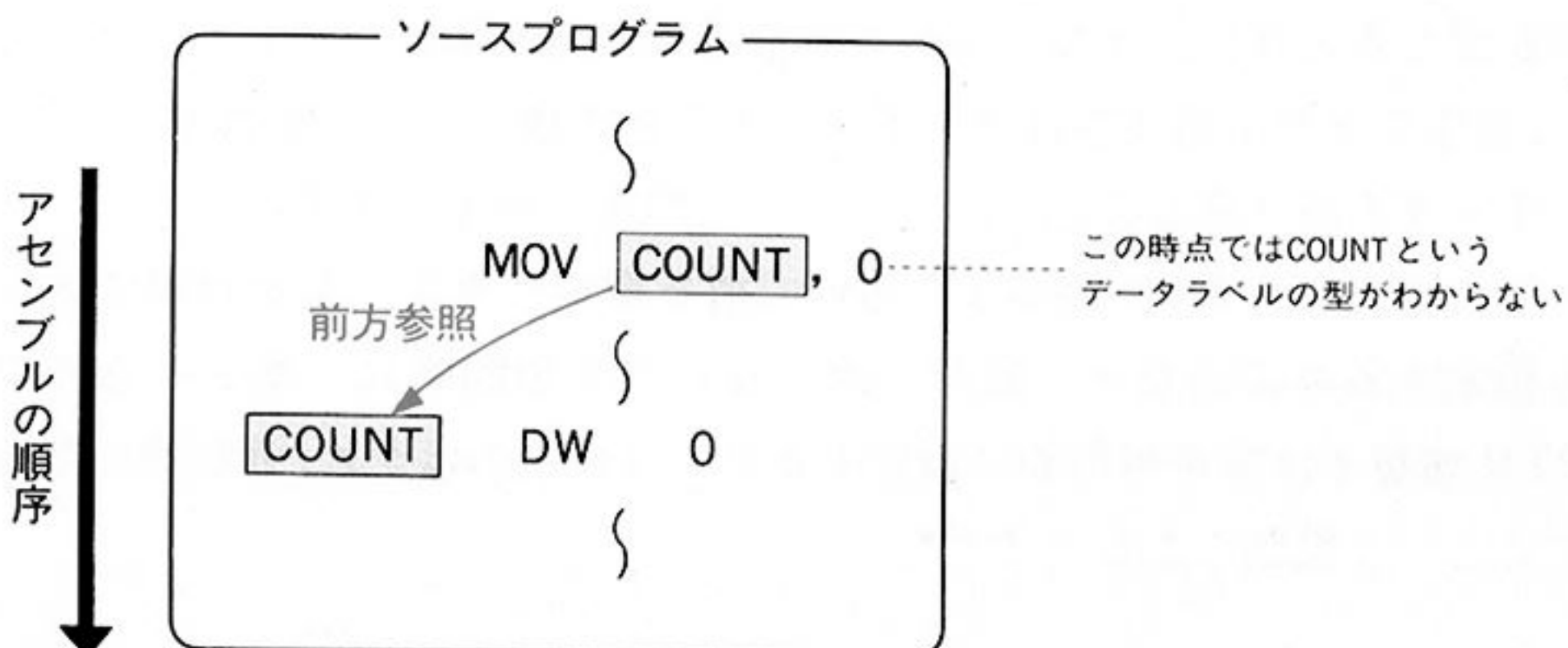



図3-13　データラベルを前方参照している場合

WORD型に対する命令よりもバイト数が短いので，MASMは命令のバイト数を調整するため余分なNOP命令(103ページのコラム参照)を入れて以降のアドレスがずれないようにします．

このNOP命令はプログラムの実行にはほとんど影響はありませんが，プログラムが多少大きくなってしまいます．したがってデータレベルを前方参照する場合には，PTR演算子を用いてはっきりと型を指定しておく方が望ましいでしょう．

データ型の強制的な変更

データラベルの定義と違う型でアクセスする場合にもPTR演算子を使います．たとえば，データラベルを「STR　DB　'漢字&ABC'」と定義してあるとします．このデータラベルを次のような命令でアクセスするとどうなるでしょうか．

```
MOV  AX, STR [SI]
```

この命令は，SIレジスタをインデックスとしてバイト型のデータラベルからAXレジスタへ1ワードのデータを転送するという命令です．この命令は，型が一致せずエラーとなります．

バイト型で定義しておきながらワード型としてアクセスするなんて，そもそもおかしいと思うかもしれませんが，マシン語プログラムではよくあることであり，融通がきくからこそ便利なのです．たとえば，文字列データに漢字が混じるときに，アルファベットなどの半角文字はバイトデータとして扱い，漢字などの全角文字はワードデータとして扱うという場合や，グラフィックデータを扱う場合などに，こういった問題は発生します．

この場合にもPTR演算子を用いて型を指定します．型が判別できないから指定するのではなく，型が一致しないので強制的に一致させるのです．PTR演算子は型を明示的に指定するという役割のほかに，強制的に型を変更するという役割もあるのです*．

---

*PTR演算子にはここに挙げた以外の使い方もあるが，それについては5章で解説する．

3.7 SEGMENT擬似命令とASSUME擬似命令

最後に残ったのがSEGMENT擬似命令とASSUME擬似命令です．この2つの擬似命令はセグメントに関する擬似命令なので，くわしくは4章で解説することにして，ここではごく簡単に説明するにとどめます．

SEGMENT〜ENDS擬似命令

［書式］ セグメント名　SEGMENT
　　　　　　　　⋮
　　　　セグメント名　ENDS

・セグメント名は<アルファベット，@，$，_，?，数字>からなる文字列で，数字で始まることはできない．

SEGMENT擬似命令は図3-14に示すように，ENDS擬似命令と対を成しています．つまり，SEGMENT擬似命令とENDS擬似命令は合わせて1つの擬似命令であり，その間に書いた部分を囲むことになります．

```
           ASSUME   CS:CODE,DS:CODE
CODE       SEGMENT

           ORG      100H
START:     MOV      BX,0
NOINPUT:MOV         AH,06H
   ～～～～～～～～～～～～～～～～～～～～～～～～～
KNOTI      DB       'KUMORI NOTI AME ...

TABLE      DW       OFFSET HARE,OFFSET KUMORI,OFFSET AME
           DW       OFFSET ANOTI,OFFSET KNOTI

CODE       ENDS
           END      START
```

SEGMENT〜ENDS擬似命令で囲まれた部分は1つのセグメントになる

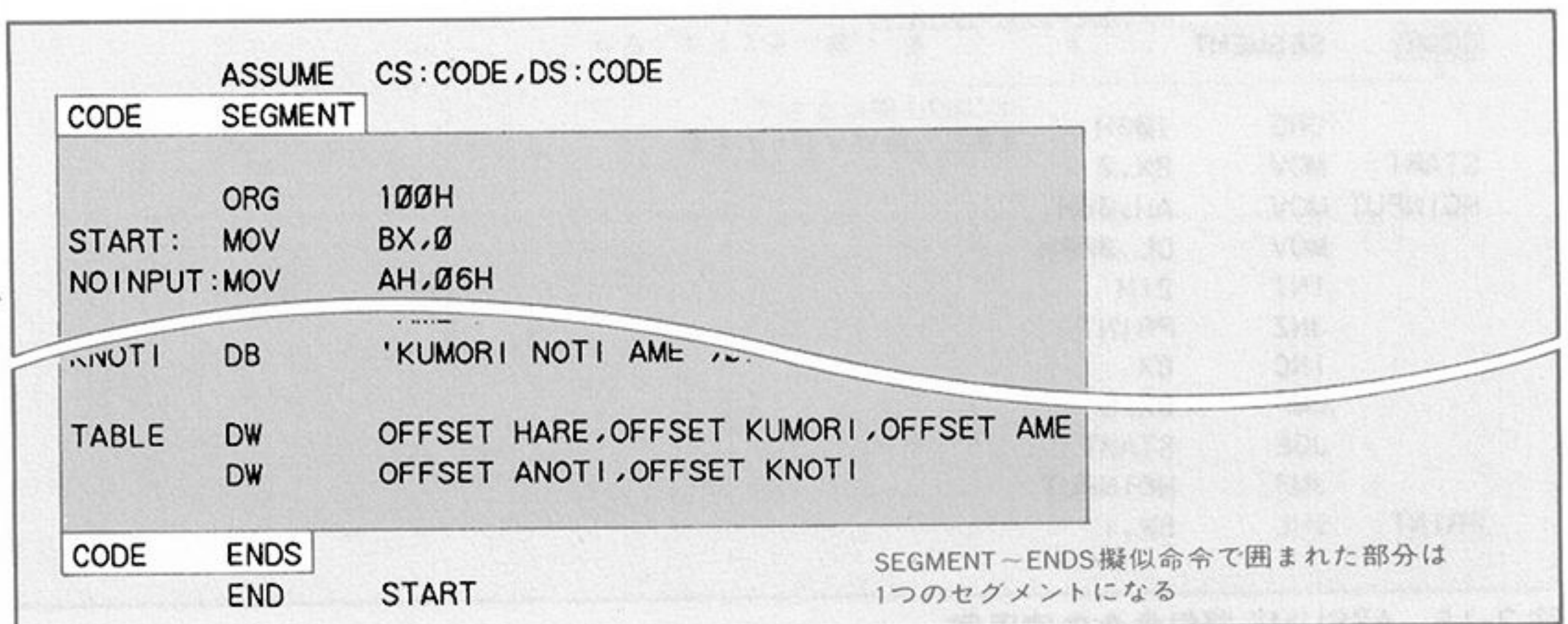



図3-14　SEGMENT〜ENDS擬似命令の使用例

SEGMENT～ENDS擬似命令は，囲まれた部分の内容を1つのセグメントとして定義します．SEGMENTおよびENDS擬似命令の前に付けられた名前が，セグメントに付けた名前を表します．セグメントの名前はラベル同様好きな名前を付けることができます．COMモデルのプログラムでは，セグメントは1つでなければならないのでEND擬似命令を除くプログラム全体をSEGMENT～ENDS擬似命令で囲みます．

本章ではCOMモデルのプログラムは，全体を1つのセグメントとして定義しなければならないということを覚えておいてください．

ASSUME擬似命令

［書式］ ASSUME　セグメントレジスタ名：セグメント名

ASSUME擬似命令は，セグメントとセグメントレジスタの対応をアセンブラに指示するための擬似命令です．くわしくは4章で解説します．

ここでは，セグメントに付けた名前を使って図3-15のように指定すればよいことだけを覚えておいてください．COMモデルのプログラムを作成するだけなら，SEGMENTおよびASSUME擬似命令については，その本質を知る必要はなく，この通りに書けばよいことさえ知っていれば十分です．

```
        ASSUME  CS:CODE,DS:CODE
CODE    SEGMENT

        ORG     100H
START:  MOV     BX,0
NOINPUT:MOV     AH,06H
        MOV     DL,0FFH
        INT     21H
        JNZ     PRINT
        INC     BX
        CMP     BX,5
        JGE     START
        JMP     NOINPUT
PRINT:  SHL     BX,1
```

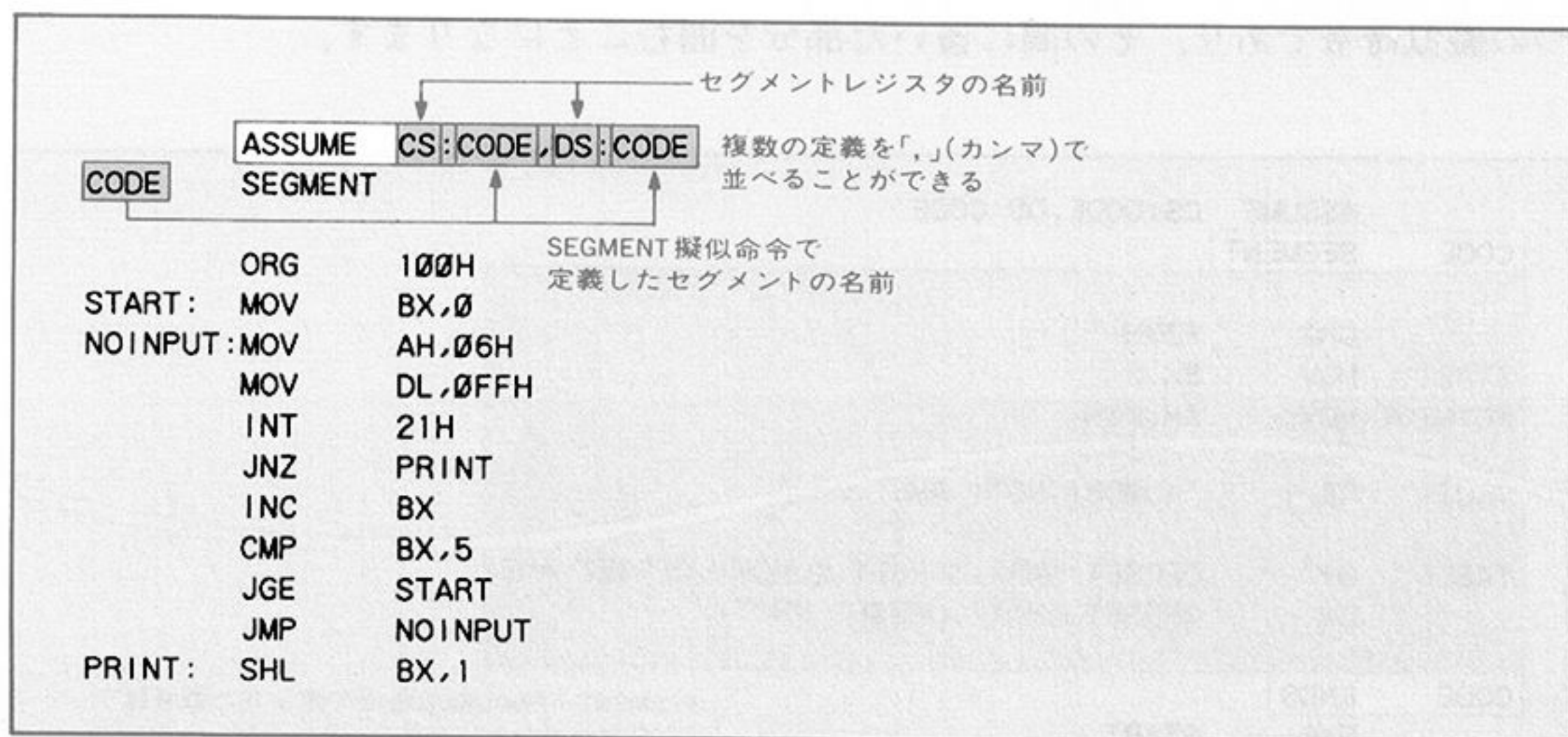



図3-15　ASSUME擬似命令の使用例

3.8

読みやすいプログラムを書くために

前節でCOMモデルのプログラムを作成するために必要な擬似命令はすべて解説しました．本節では，少しわき道にそれてプログラムの体裁について説明していきます．

プログラムはただ書けばよいというものではありません．うまく動かなかったり，後で改良したくなったときに必ず読み返すものです．そのとき，一定の約束にしたがった体裁で読みやすく書いておけば，プログラム開発の効率が上がることはいうまでもないでしょう．〝プログラムは文書でもある〟と思って，極力読みやすく理解しやすいプログラムを書くように心がけることが大切です．

フィールド

アセンブリ言語のプログラムは，擬似命令とマシン語命令から構成されます．各命令は1行に1つずつ書きます．各命令の書式にしたがっていればどのように書いてもよいのですが，次の図3-16に示すような決まったフィールドに書くのが普通です．

図3-16 ソースプログラムのフィールド

左側のフィールドは「ラベルフィールド」で，ラベル名やセグメント名を書きます．中央のフィールドは「命令フィールド」で，各種擬似命令やマシン語命令を書きます．右側のフィールドは「コメントフィールド」で，後で解説するコメントを書きます．

各フィールドの先頭はタブ位置に揃えます．タブ位置は，通常8桁ごとに設定されています．いくつ目のタブ位置に揃えるかは各人の好みによります．長いラベル名を使いたい人は命令フィールドをタブ2の位置に揃えてもよいでしょう．

フィールドは厳密なものではなく，はみ出しても一向にかまいません．たとえば，ラベルや命令が長くなって次のフィールドにはみ出してしまうことがありますが，無理に短くする必要はありません．そのかわり，ラベルと同じ行に命令を書かずに次の行にずらすなど，読みやすくなるような工夫をすればよいでしょう．

コメント

［書式］ ; 任意の文章

コメントとは注釈のことで，プログラムの中に混ぜておく解説の文章を指します．たとえば，図3-16のように各命令のプログラム中における意味を短い文章で書いておきます．

コメントの有無は，プログラムの読みやすさに大きく影響します．プログラム中にコメントを書いておくことでプログラムを理解しやすいものにすることができます．後で読み返してもすぐに理解できるプログラムにするためには，適切なコメントをできるだけたくさんつけておくことが大切です．読者のみなさんもプログラムを書くときには，必ずコメントをつけるようにしてください．

コメントは，「;」（セミコロン）で始まる文章です．プログラムの中に「;」が現れると，そこからその行の終わりまではすべてコメントとみなされます．コメントには何でも書くことができ，日本語の文章を書くこともできます．

コメントはラベルや命令と同じ行に書くことも可能です．このため図3-

17のように命令フィールドの右側をコメントフィールドとして利用します．アセンブラでのプログラミングに慣れないうちは，1つの命令ごとにコメントを書いておくとよいでしょう．

コメントはコメントフィールドだけでなく，行の先頭から書いてもかまいません．マシン語のプログラムは，いくつかの命令を単位として1つの機能を達成しています．その単位ごとに何を行っている部分なのかを解説するコメントをつけるのも1つの方法です．たとえば，次の図3-17のようになります．

図に示したようにプログラムの先頭に，プログラムの解説や使い方を書いておくのもよい方法です．そのほか，プログラムの作成者，作成日付なども書いておくとよいでしょう．プログラムに変更を加えたときにも，その日付や変更内容などを書いておくときっと役に立ちます．

なお，本書で示すプログラム例では，コメントをあまりつけていないものを示していますが，みなさんがプログラムを作成する場合には，ぜひコメントをたくさんつけてください．

ここで紹介したプログラムの書き方は，あくまで1つのスタイルにすぎません．絶対にこのとおりに書かなければならないわけではありません．他の書籍や雑誌のプログラム例を参考にして，よいところはどんどん自分のスタイルに組み込んでいきましょう．

```
;          otenki.asm  (天気予報プログラム)
;            説明
;              当たるも八卦、当たらぬも八卦
;              結果を信じるかどうかはあなた次第です。
;            使用法
;              A>OTENKI
;              何かキーを押すと明日の天気が表示されます。
;
           ASSUME   CS:CODE,DS:CODE
CODE       SEGMENT

           ORG      100H
;          キー入力を待つ
;          その間、カウンタを0～4でインクリメントさせる
START:     MOV      BX,Ø                ;BXレジスタを0に初期化する
NOINPUT:MOV         AH,Ø6H              ;ファンクション番号6 (1文字入出力)
           MOV      DL,ØFFH             ;1文字入力を指定
           INT      21H                 ;ファンクションコールを実行
           JNZ      PRINT               ;キー入力があれば表示へ
           INC      BX                  ;カウンタを1増やす
           CMP      BX,5                ;メッセージの数5を超えたか?
           JGE      START               ;超えていれば初期化へ
           JMP      NOINPUT             ;再びキー入力へ
;
;          キー入力があったので番号に対応した
;          メッセージを表示する
PRINT:     SHL      BX,1                ;カウンタを2倍する
           MOV      DX,TABLE[BX]        ;メッセージのアドレスをDXレジスタにセット
           MOV      AH,Ø9H              ;ファンクション番号9 (文字列表示)
           INT      21H                 ;ファンクションコールを実行
           MOV      AH,4CH              ;ファンクション番号4CH (プログラム終了)
           MOV      AL,ØØH              ;リターンコード0
           INT      21H                 ;ファンクションコール実行
;
;          天気を表すメッセージ
HARE       DB       'HARE',ØDH,ØAH,'$'
KUMORI     DB       'KUMORI',ØDH,ØAH,'$'
AME        DB       'AME',ØDH,ØAH,'$'
ANOTI      DB       'AME NOTI HARE',ØDH,ØAH,'$'
KNOTI      DB       'KUMORI NOTI AME',ØDH,ØAH,'$'
;
;          メッセージの先頭アドレスの配列
TABLE      DW       OFFSET HARE,OFFSET KUMORI,OFFSET AME
           DW       OFFSET ANOTI,OFFSET KNOTI

CODE       ENDS
           END      START
```

図 3-17　ブロックごとのコメント

COMモデルのプログラム実習

これまでの解説でCOMモデルのプログラムを書くことができるようになりました．次はいよいよアセンブルして実行してみます．

アセンブルの操作

実際の操作は1章ですでに体験しているので，ここでは簡単に復習するにとどめます．COMモデルの実行ファイルを作成するには，次の3つの操作が必要です．ソースプログラムのファイル名は，OTENKI.ASMであるとします．

```
MASM  OTENKI;                      ……アセンブル
LINK  OTENKI;                      ……リンク
EXE2BIN  OTENKI  OTENKI.COM        ……ファイル形式変換
```

アセンブルとリンクの操作では，ファイル名の後に「；」(セミコロン)を付けるのを忘れないでください．なお，アセンブラやリンカのよりくわしい使い方はAPPENDIXに解説してありますから，参照してください．

アセンブル(MASMコマンド)

図3-18は本章で解説した中心の話題であるアセンブルという操作です．MASMコマンドを実行することによってアセンブルが行われ，オブジェクトファイルが作られます．アセンブルを行った後には，「.OBJ」という拡張子の付いたオブジェクトファイルができています．

```
A>MASM OTENKI;
Microsoft MACRO Assembler  Version 3.00
(C)Copyright Microsoft Corp 1981, 1983, 1984

49694 Bytes free

Warning Severe
Errors  Errors
0       0

A>
```

図 3-18 アセンブル

アセンブル時には，エラーが発生する場合があります．ソースプログラムに文法的な誤りがあると，アセンブラは図 3-19 のようにエラーメッセージを出力します．

```
A>MASM OTENKI;
Microsoft MACRO Assembler  Version 3.00
(C)Copyright Microsoft Corp 1981, 1983, 1984

0113 E4                          PRINT:  SHL     BX.1
E r r o r   ---        10:Syntax error
                                           エラーメッセージ
49694 Bytes free

Warning Severe
Errors  Errors  エラーが1個あったことを示している
0       1

A>
```

図 3-19 アセンブルエラー

アセンブルエラーが発生するとアセンブルは正常に行われていないわけですから，次のステップに進むことはできません．エラーの原因を探してソースプログラムを修正しなければなりません．

エラーがどこで起こったのかを知るために，アセンブルリストファイルを作成します．アセンブルリストファイルを作成するには，次のように MASM コマンドを起動します．

MASM OTENKI, , OTENKI;

「,」(カンマ)を2つ並べて，もう一度ソースファイル名を書くことにより，MASMにリストファイルを出力するように指定したことになります．このコマンドによって図3-20のようにリストファイルが作成されます．

リストファイルは，アセンブルの結果を細かく報告するために出力されるファイルです．アセンブルの結果，出力されるマシン語コードや，アドレスなどの情報がソースプログラムに対応する形で出力されます．

```
                    ┌─アセンブルリストファイルの出力を指定
A>MASM OTENKI,,OTENKI;   ……アセンブルリストファイルは，拡張子を省略すると
Microsoft MACRO Assembler  Version 3.00      「LST」が自動的に指定される
(C)Copyright Microsoft Corp 1981, 1983, 1984

 0113  E4                   PRINT:  SHL     BX.1
 Error    ---          10:Syntax error

49694 Bytes free

Warning Severe
Errors  Errors
0       1

A>TYPE OTENKI.LST
 Microsoft MACRO Assembler  Version 3.00                 Page    1-1
                                                               05-15-88

                                          ASSUME  CS:CODE,DS:CODE
 0000                             CODE    SEGMENT

 0100                                     ORG     100H
 0100   BB 0000                   START:  MOV     BX,0
 0103   B4 06                     NOINPUT:MOV     AH,06H
 0105   B2 FF                             MOV     DL,0FFH
 0107   CD 21                             INT     21H
 0109   75 08                             JNZ     PRINT
 010B   43                                INC     BX
 010C   83 FB 05                          CMP     BX,5
 010F   7D EF                             JGE     START
 0111   EB F0                             JMP     NOINPUT
 0113   E4                        PRINT:  SHL     BX.1     この行にエラーがあった
 Error    ---          10:Syntax error
 0114   8B 97 015A R                      MOV     DX,TABLE[BX]
 0118   B4 09                             MOV     AH,09H
 011A   CD 21                             INT     21H
```

図3-20 アセンブルリスト

エラーが発生した場合，リストファイルにはアセンブルの際に出力されたのと同じエラーメッセージがソースファイル中にも出力されています．つまり，その行に文法的な誤りがあるということがわかるわけです．

この例では，「,」(カンマ)と「.」(ピリオド)を間違えていました．そこで，ソースファイルのミスを修正して再びアセンブルします．エラーが起こらなくなるまでこの操作を繰り返し，エラーがなくなったら次のステップへ進むことができます．

COLUMN

MASMのバージョンによるエラーメッセージの違い

本文では，MASMのVer3.0を使用した場合を示していますが，MASMはさらに新しいバージョンが発売されています．Ver4.0以降のMASMでは，エラーメッセージの表示方法が異なっています．たとえば，図3-19のようなエラーの場合，次の図のように表示されます．

```
A>MASM OTENKI;
Microsoft (R) Macro Assembler  Version 4.00
Copyright (C) Microsoft Corp 1981, 1983, 1984, 1985.  All rights reserved.

OTENKI.ASM(14) : error 10: Syntax error

  50258 Bytes symbol space free

      0 Warning Errors
      1 Severe  Errors

A>
```

エラーメッセージは次のような書式で表示されています．

ソースファイル名(行番号)：error　エラー番号：　エラーの種類

エラーメッセージのなかに行番号が含まれているので，わざわざリストファイルを出力しなくてもエラーの発生した場所がわかります．

リンク(LINK コマンド)

図 3-21 の操作はリンクと呼ばれる操作です．リンクの本当の意味は 5 章で解説しますから，ここではオブジェクトファイルを EXE 型の実行ファイルに変換するコマンドだと思っていてもかまいません．

```
A>LINK OTENKI;↵

Microsoft 8Ø86 Object Linker
Version 3.Ø1 (C) Copyright Microsoft Corp 1983, 1984, 1985

Warning: no stack segment    警告メッセージが出力される

A>
```

図 3-21　リンク

アセンブルの結果作られたオブジェクトファイルは，そのままでは実行できません．オブジェクトファイルにはマシン語コードのほかに，分割アセンブルを可能にするためのさまざまな情報が含まれているからです(くわしくは 5 章で解説)．実行可能なファイルとするためリンクの操作を行い，これらの情報を確定しなければなりません．

リンクの操作を行うと図 3-21 のように，Warning(警告)メッセージが出力されます．このメッセージは「このプログラムにはスタック領域が用意されていない」と警告しています．しかし，COM モデルのプログラムではスタック領域は起動時に自動的に設定されるので，わざと設定していないのです．したがって COM モデルのプログラムをリンクするときには，この警告メッセージは無視します．

ファイル変換(EXE2BINコマンド)

リンクの操作によって「.EXE」の拡張子を持ったファイルができます.EXE型の拡張子を持ったファイルは,コマンドとして実行することができるはずですが,本章のプログラムはCOMモデルとして作成してあるので,COM形式に変換しない限り実行はできません.誤って実行してしまうと,暴走する恐れがあって危険なので,決して実行しないでください.

EXE2BINコマンドはEXE形式の実行ファイルをCOM形式の実行ファイルに変換するためのものです.COM形式として作成したプログラムは,必ずこのコマンドを使ってCOM形式に変換しなければなりません(図3-22).

なお,COM形式とEXE形式の違いについては4章でくわしく解説します.

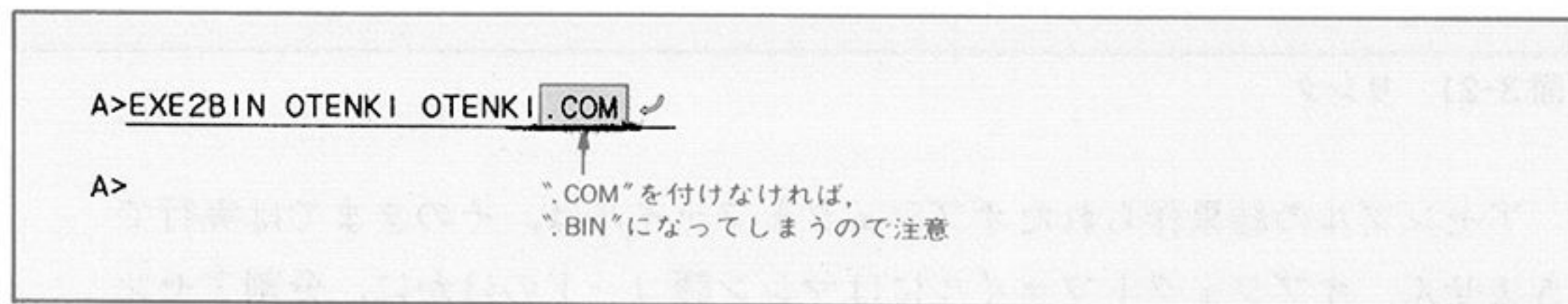



図3-22 ファイル変換

デバッガ(SYMDEBコマンド)

ここまでの3つの操作で,ソースファイルをアセンブルし実行ファイルを作成することができました.本章で学習するべきことはこれですべて終了したわけですが,理解を深めるために擬似命令の効果を実際に確認してみましょう.

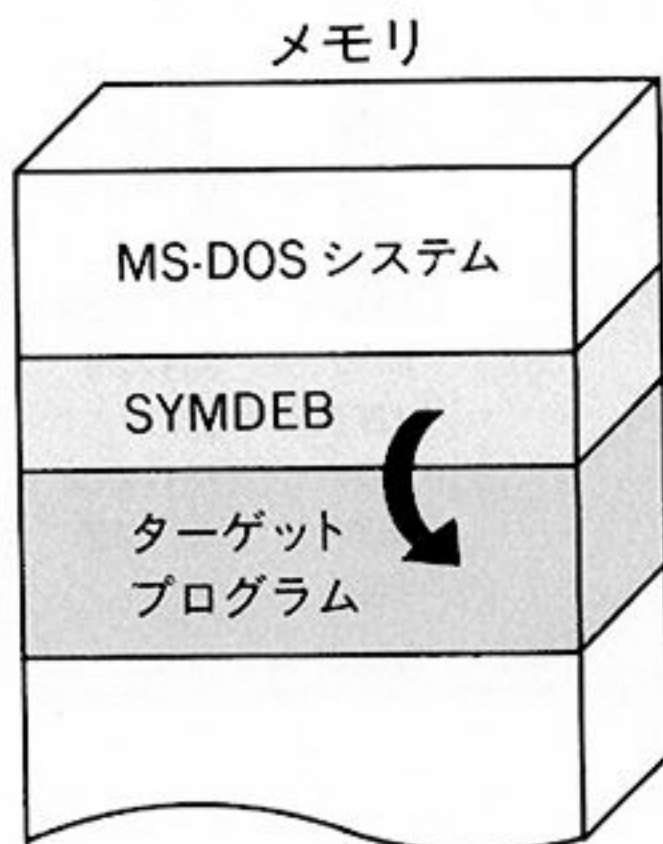
そこで,できあがった実行ファイルを覗いてみることにします.そのためにはSYMDEBコマンドが力を発揮します.SYMDEBはデバッガと呼ばれるように,本来はプログラムのデバッグ,つまり思いどおりに動かないプログラムの動作をチェックするためのツールです.本書ではデバッグの方法までは解説しませんが,SYMDEBを使ってプログラムの動作を確かめていくことにします.

では，できた実行ファイルをSYMDEBの上で実行して，アセンブラの仕事を再確認してみましょう．まず，SYMDEBを次のようにして起動します．

```
SYMDEB OTENKI.COM
```

パラメータとして指定するのは，作成したCOM形式の実行ファイルです．このファイルはデバッグの対象となるプログラムですから，「ターゲットプログラム」と呼ばれます．SYMDEBは図3-23に示すように，SYMDEBそのものがロードされた直後のメモリにターゲットプログラムをロードします．SYMDEBはターゲットプログラムがMS-DOSによってロードされ実行されるのとまったく同じ状況を用意します．そのまま実行すれば，MS-DOSから直接実行したのと同じ結果が得られます．

SYMDEBはターゲットプログラムの実行を監視して，あらかじめ指定しておいたアドレスで実行を停止したり，1命令ずつ実行させることができます．マシン語プログラムのコード部分を逆アセンブルしたり，データ部分の内容を表示させたり変更したりすることもできます．これらの機能は主としてプログラムのデバッグのために用意されているもので，プログラムの実行を逐一観察しその動作を確かめることができます．

本節ではこれらの機能をアセンブラの役割を確かめるために使用します．ソースプログラムに記述した擬似命令がどのような効果をもたらしているか

●SYMDEBは，ターゲットプログラムがコマンドラインから起動されたのと同じ状況を用意する．

●ターゲットプログラムはSYMDEBの管理下に置かれる．SYMDEBのコマンドによってターゲットプログラムの中身を覗いたり，変更することができる．部分的に実行することなども可能．

図3-23 デバッガの仕組み

を，オブジェクトプログラムを覗いてみることによって確かめます．もちろん，本書で解説した以外の擬似命令の効果を試してみることも同様にできますので，各自でやってみてください．もっとも，それよりもプログラムがうまく動いてくれずに，プログラムの動作を観察してその原因を探るためにSYMDEB*を使うことが多いとは思いますが．

ラベルの効果

ソースプログラム中のラベルは，オブジェクトプログラムにはどういう形で反映されているでしょうか．図 3-24 はソースプログラムとオブジェクトプログラムを対比させたものです．ラベルの部分が正しく対応するアドレスに変換されていることがわかると思います．

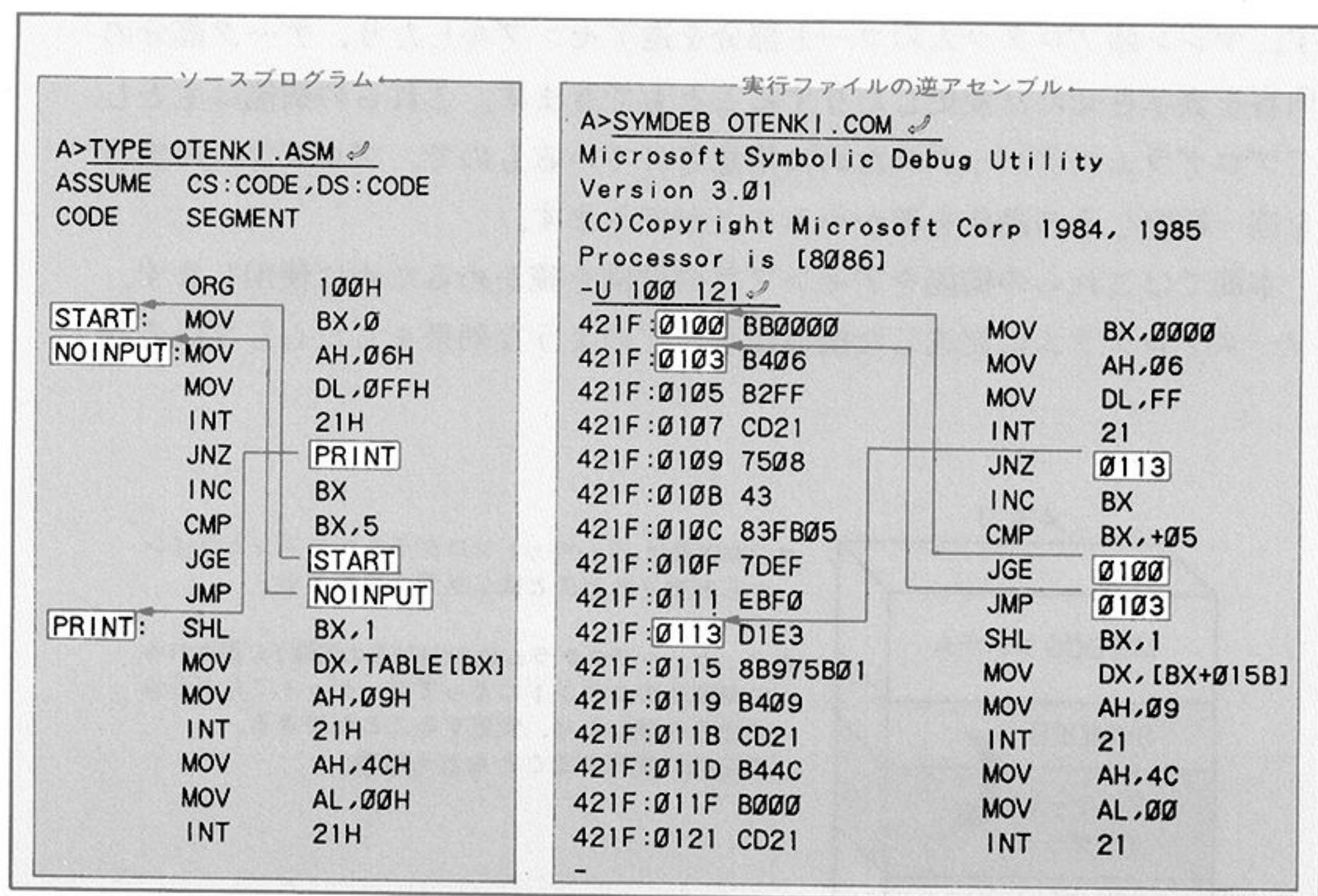



図 3-24　ラベルの効果

---

* MASM バージョン 2.15 以前のシステムには，SYMDEB コマンドの代わりに DEBUG コマンドが付属している．本章で解説する範囲では，まったく同じように利用することができる．

ジャンプ先のアドレスはプログラムの変更によって変わる可能性があります．今，わざとNOP命令をプログラムに挿入してどのように変わるかを見てみましょう（図3-25）．

ソースファイル

```
A>TYPE OTENKI.ASM
         ASSUME    CS:CODE,DS:CODE
CODE     SEGMENT

         ORG       1ØØH
START:   MOV       BX,Ø
NOINPUT:MOV        AH,Ø6H
         MOV       DL,ØFFH
         INT       21H
         JNZ       PRINT
         NOP
         INC       BX
         CMP       BX,5
         JGE       START
         JMP       NOINPUT
PRINT:   SHL       BX,1
         MOV       DX,TABLE[BX]
         MOV       AH,Ø9H
         INT       21H
         MOV       AH,4CH
                           ; L
         MOV       AL,ØØH
```

NOP命令を挿入した

実行ファイルの逆アセンブル

```
A>SYMDEB OTENKI.COM
Microsoft Symbolic Debug Utility
Version 3.Ø1
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8Ø86]
-U 1ØØ 122
421F:Ø1ØØ BBØØØØ          MOV    BX,ØØØØ
421F:Ø1Ø3 B4Ø6            MOV    AH,Ø6
421F:Ø1Ø5 B2FF            MOV    DL,FF
421F:Ø1Ø7 CD21            INT    21
421F:Ø1Ø9 75Ø9            JNZ    Ø114
421F:Ø1ØB 9Ø              NOP
421F:Ø1ØC 43              INC    BX
421F:Ø1ØD 83FBØ5          CMP    BX,+Ø5
421F:Ø11Ø 7DEE            JGE    Ø1ØØ
421F:Ø112 EBEF            JMP    Ø1Ø3
421F:Ø114 D1E3            SHL    BX,1
421F:Ø116 8B975CØ1        MOV    DX,[BX+Ø15C]
421F:Ø11A B4Ø9            MOV    AH,Ø9
421F:Ø11C CD21            INT    21
421F:Ø11E B44C            MOV    AH,4C

421F:Ø12Ø BØØØ            MOV    AL,ØØ
```

アドレスが1バイト分ずれている

図3-25 ラベルに対応するアドレスの変化

前ページの図3-24と図3-25と比べてみてください．ソースプログラムの上ではNOP命令を挿入したにすぎません．しかしオブジェクトプログラムの方は，アドレスと命令の対応が異なる部分があります．1バイトのマシン語に相当するNOP命令を挿入したことにより，それ以降の命令のアドレスがすべて1バイトずつずれています．したがってジャンプ命令の飛び先のアドレスが1バイト分ずれています．

このことから，アセンブラが自動的にアドレスを計算してくれていることによる効果がよくわかります．もしもアセンブラにラベルの機能がなく，アドレスを直接指定しなければならないとしたら，どんなにたいへんかは想像に難くないでしょう．

ORG 擬似命令の効果

ORG 擬似命令の効果を確かめるために，ORG 擬似命令なしでアセンブルするとどうなるかを試してみましょう．図 3-26 に ORG 擬似命令を取り除いたソースプログラムから作成した実行プログラムを SYMDEB でロードした様子を示します．

```
421B:010C 83FB05       CMP    BX,+05
421B:010F 7DEF         JGE    0100
421B:0111 EBF0         JMP    0103    015Bから変化した(図3-24を参照)
421B:0113 D1E3         SHL    BX,1
421B:0115 8B975B00     MOV    DX,[BX+005B]
421B:0119 B409         MOV    AH,09
421B:011B CD21         INT    21
```

図 3-26　ORG 擬似命令を使用しない場合

オブジェクトプログラムをよくみると，データラベルに対応するアドレスの値が違うものになっていることがわかります．$015B_H$であるはずが，ちょうど 0100_Hだけずれて $005B_H$になっています．プログラムはアドレス 0100_Hからのメモリにロードされていますが，0000_Hから始まるものとしてアセンブルされたことがわかります．

なお，ジャンプ命令の飛び先アドレス部分は変化していません．これはジャンプ命令の飛び先アドレスは，相対アドレス*で表されるからです．相対アドレスはそのアドレスからどれだけ離れているかを表すので，異なるアドレスにロードされても飛び先との関係は保たれます．

DB 擬似命令の効果

DB 擬似命令などのデータ定義擬似命令については，その効果も含めて 74 ページの表 3-1 で解説しています．具体的な効果のよくわからないものがあれば，自分で SYMDEB を使って確かめてみるとよいでしょう．

---

*前書「はじめて読む 8086」を参照のこと

COLUMN

NOP命令
―何もしない命令？―

本文で，マシン語命令を１バイト増やすためにNOP命令を利用しました．NOPはNo OPerationの略で，「何もしない」という意味です．その名の通り，NOP命令は実行しても何もしませんが，正確には「何もしないことを実行する」命令と考えなければなりません．マシン語命令の１つであるからには，CPUによって読み込まれ，解釈され，その結果が実行されます．つまり，「何もしないで次の命令に進む」ことを実行するのです．この考え方は重要で，NOP命令を実行するにも時間がかかるということを意味します．

CPUが周辺機器と入出力を行う場合，一般にCPUの速度は周辺機器よりも速いのでその速度差が問題になることがあります．たとえば，CPUから周辺機器へいくつかの指令を出力する場合を考えます．周辺機器へ指令を出力した直後に次の指令を出力しようとしても，周辺機器は前の指令を取り込んで解釈している最中で次の指令を受け取れない場合があるのです．このような場合にNOP命令が使われます．最初の指令を出力した後，適当な数のNOP命令を実行し時間をかせいでから次の指令を出力するのです．

4

セグメントの本格的活用

8086CPU ではセグメント方式と呼ばれるメモリ管理方式を採用しています．セグメント方式は 8086CPU の大きな特徴の 1 つであり，MASM にもこのセグメントを扱うための機能が豊富に用意されています．

3 章で解説したように，MASM では簡単なプログラムでもセグメントに関する指定を行わなければなりません．セグメントについてあまり理解していない段階では，これらの指定は面倒なだけでありプログラミングを難しいものにしているように思えるでしょう．しかし，セグメントを自在に扱えることが MASM の大きな特色であり，理解してしまえばそれほど難しいものではないのです．

セグメント方式を 64K バイトを超えるメモリを扱うための間に合わせ的な手段と考えてしまうと，セグメント方式の価値は半減し，プログラミングも難しく感じてしまいます．セグメント方式は大容量のメモリをメモリブロック(セグメント)の集まりとして扱う手法であるという点に注目しましょう．8086CPU のメモリ空間を 1M(メガ)バイトの連続した空間ではなく，いくつもの小さなメモリブロックから構成されるものとして考えるのです．

本章ではこの考え方をわかりやすく解説していきます．8086CPU，すなわち MASM におけるセグメントの意味をよく理解してください．

4.1
セグメントの概念

セグメントに関する知識の必要性

セグメントの概念そのものについての解説を始める前に，まず例題のプログラム，ESC.COMを**リスト4-1**に紹介します．このプログラムを通して，MS-DOSで(すなわち8086CPU用の)プログラムを作成するためには，セグメントの概念は避けて通れないものであり，ぜひ理解しておくべきであることを示していきます．

簡単に解説すると，ESC.COMはエディタなどでは入力しにくいエスケープシーケンスを扱いやすくするプログラムで，表示される文字を反転状態にするエスケープシーケンスをサポートします．その使い方はまず，**図4-1**のように反転させたい部分を〝[ ]″で囲んだファイルを用意します．そして，MS-DOSのリダイレクト機能を使ってESC.COMの入力をそのファイルに切り替え，出力を目的のファイルに切り替えます*．するとできあがったファイルでは〝[　]″で囲んだ部分の前後に，表示を反転させるエスケープシーケンスと，通常表示に戻すエスケープシーケンスが挿入されています**．このファイルを表示すると，その部分が反転表示されます．

このプログラムでは，文字をコンソール(リダイレクトされた場合はファイル)から1文字ずつ読み込んできては〝[″もしくは〝]″かどうかをチェックし，やはり1文字ずつコンソール(ファイル)に出力しています．実はこの方法には欠点があります．それは入力ファイルが大きくなると，実行完了までにかなりの時間がかかることです．それは，1文字ごとにMS-DOSを呼び出して入出力を行っていることが原因です．

---

*このように，入力ファイルと出力ファイルが1つずつあり，入力したファイルの一部を加工して出力するプログラムを「フィルタ」と呼ぶ．
**〝[″または〝]″そのものを表示することはできない．また，漢字コードの2バイト目がこれらの文字のコードと一致する漢字は使用できない．

リスト4-1　エスケープシーケンスジェネレータ ESC.ASM

```
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,ES:CODE

        ORG     100H
START:
M_LOOP:
;-- get char -- ……………1文字読みだしルーチン
        MOV     AH,8
        INT     21H                                         入力
;-- get char end --

ESC:
;-- transfer routine --
        CMP     ESCFLAG,BYTE PTR 1 ……………反転中かどうかを調べる
        JE      NORMCHK
        CMP     AL,'[' ……………反転を開始するかどうかを調べる
        JNE     THROUGH
;-- start reverse output -- ……………反転開始
        MOV     ESCFLAG,BYTE PTR 1
        MOV     BX,OFFSET REVSTR
        MOV     CX,4
        JMP     PUTS
NORMCHK:
;-- check reverse end ? --
        CMP     AL,']' ……………反転終了かどうかを調べる          変換処理
        JNE     THROUGH
;-- end reverse output -- ……………通常表示に戻す
        MOV     ESCFLAG,BYTE PTR Ø
        MOV     BX,OFFSET NORMSTR
        MOV     CX,4
        JMP     PUTS
THROUGH:
;-- output through -- ……………文字をそのまま出力する
        MOV     CHRBUF,AL
        MOV     BX,OFFSET CHRBUF
        MOV     CX,1
;-- escape sequence insertion end --

;-- put char -- ……………変換結果の出力
PUTS:
        MOV     DL,[BX]
        INC     BX
        MOV     AH,2                                        出力
        INT     21H
        LOOP    PUTS

        CMP     DL,'Z'-'A'+1 ……………^Zならば終了
        JNE     M_LOOP                                      終了判定
```

```
;-- end of program --
        MOV     AH,4CH                              終了
        MOV     AL,Ø
        INT     21H

CHRBUF  DB      ? ……………通常出力用の仮バッファ
ESCFLAG DB      Ø ……………反転中かどうかを示すフラグ
REVSTR  DB      1BH,'[7m' …………表示を反転させるエスケープシーケンス
NORMSTR DB      1BH,'[Øm' ……………表示を通常に戻すエスケープシーケンス

CODE    ENDS

        END START
```

```
A>TYPE ASM ↵
                                          ┌──強調したい文字列の前後を"[ ]"で囲む
echo MASM start!                          ↓
echo MASM translates [assembly-language] source code into relocatable object.
echo And, MASM does [two passes] to translate the assembly-language.
masm %1;        ┐
link %1;        │COMモデル用
exe2bin %1 %1.com;┘

A>ESC < ASM > ASM.BAT ↵ ……………ESCコマンドを使ってエスケープシーケンスを埋め込む*

A>TYPE ASM.BAT ↵
                                          ┌──反転表示される
echo MASM start!                          ↓
echo MASM translates assembly-language source code into relocatable object.
echo And, MASM does two passes to translate the assembly-language.
masm %1;
link %1;
exe2bin %1 %1.com;

A>          *COMモデルのプログラムをアセンブル&リンクするためのバッチファイルを作成している
```

図 4-1 ESCコマンドの使用例

これを解決して高速の入出力を実現する方法は，入力および出力をそれぞれ「バッファリング」することです．すなわち，一度に多くの文字を読み込み，まとめて処理したのち，一度に出力するという方法です．この場合，なるべく多くの文字を一度に読み込んだ方が速度は速くなります．そのためには大きなバッファ，つまり多くのメモリ領域を必要とします．

現実のプログラム開発では，このようにできるだけ多くのメモリを扱いたい場合がよくあります．しかし，ここでぶつかるのが8086CPUに特有の「セグメントの壁」です．3章で解説したCOMモデルのプログラムではセグメントの概念を意識する必要がない代わりに，スタック領域などを含めて64Kバイトまでしか扱うことができません．64Kバイト以上のメモリを扱いたい場合には，どうしてもセグメントについての知識が必要です*．

セグメントの概念は多くのメモリを扱うための手段であり，8086CPUでプログラムを組むためには必要不可欠な知識です．本章ではセグメントの概念をくわしく解説していきます．そのうえで例題のプログラムの改良に挑戦しましょう．

利用可能なメモリ容量

私たちがプログラムを作成する場合にどれだけのメモリを利用することができるかを示すために，MS-DOSにおけるメモリ管理方法を解説します．8086CPUは1M(メガ)バイトのメモリ空間を扱うことができますが，これをMS-DOSでは図4-2のように利用しています．

図4-2の上に示されるように1Mバイトのメモリ空間のうち低位(アドレスの低い方)の部分は，MS-DOSシステムのプログラムやワークエリアとして使われます．そして，高位の部分はROM BIOSやVRAM(ビデオRAM)が割り当てられています．残った中間のRAMエリアがコマンドやアプリケーションをロードして実行するためのユーザー領域となります．つまり，この領域を私たちの作成するプログラムで利用できるわけです．

図4-2の下に示すように，使用可能メモリ，つまりユーザー領域の大きさは，COMモデルのプログラムで利用できる64K(65536)バイトよりもずっと大きいことがわかります．私たちがプログラムを作成する場合にも，これだけのメモリを利用することができるはずなのです．

*C言語などの高級言語を使っている場合にも，大量のデータを扱うプログラムではセグメントの知識が必要になることがある．スモールモデルやラージモデルといったメモリモデルの選択は，セグメントの概念と密接に関連している．

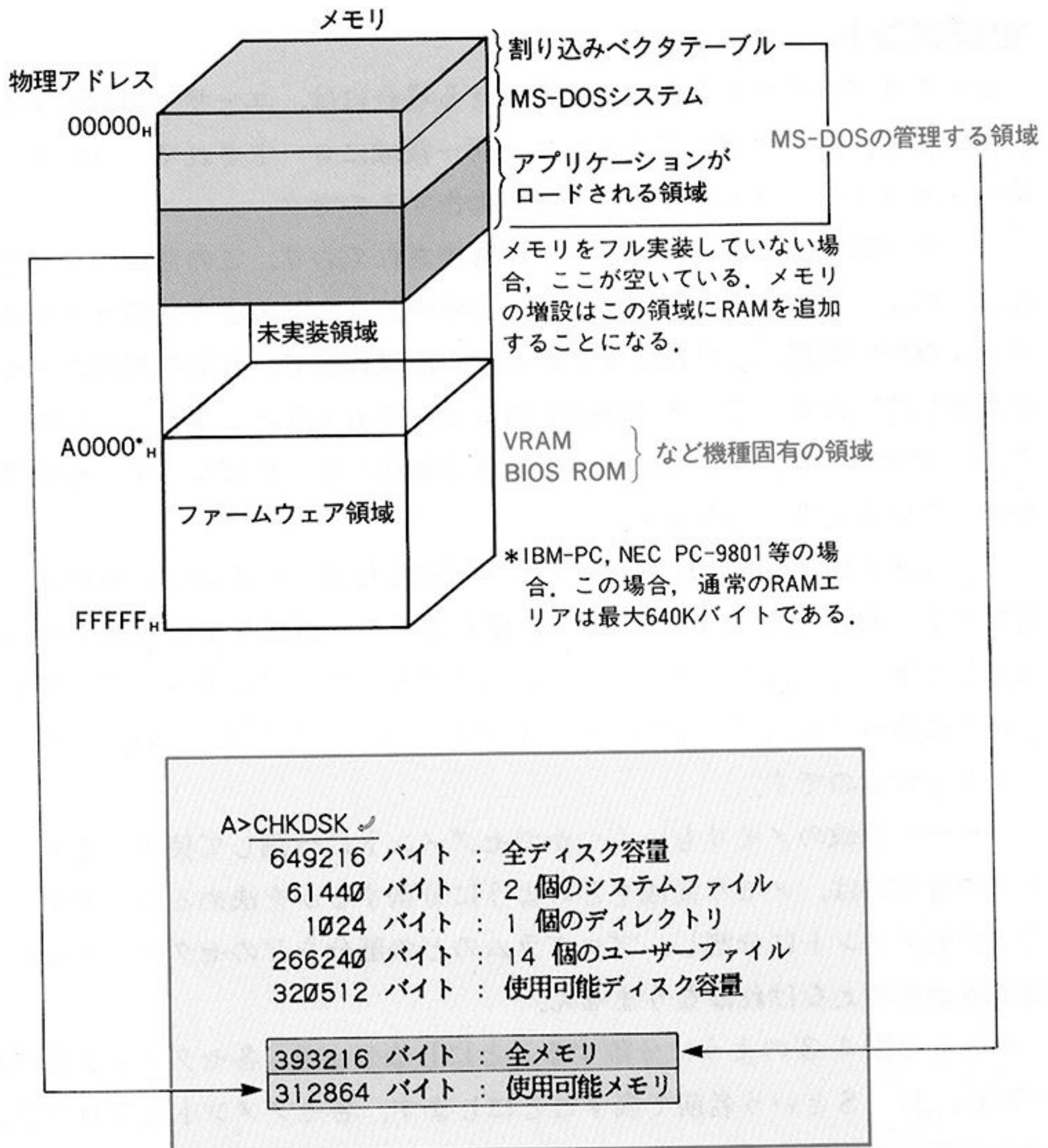



図 4-2 MS-DOS でのメモリ空間の割り当て

セグメント

私たちがプログラムを作って実行させる場合には，ユーザー領域のメモリを使います．作ったプログラムはユーザー領域にロードされて，ユーザー領域のメモリをワークエリアとして使い動作するのです．

ユーザー領域はまるごとユーザーに解放されており，この領域のなかであればどのように使うことも可能です．そのかわり，どのように使うかをユーザーが自分で管理しなければなりません．高級言語では言語処理系がメモリを管理してくれるので，その詳細を知る必要はあまりありません．しかし，アセンブラのプログラムではすべて自分で管理しなければならず，その方法を知っている必要があります．

その方法とは，8086CPUのハードウェア的な仕組みに直結した**セグメント方式**です．8086CPUは64Kバイトを超えるメモリ領域を連続した1つの領域として扱うことはできず，いくつかのメモリブロックに分割して使用するという仕組みになっています．その1つ1つのメモリブロックのことをセグメントと呼ぶのです．

ユーザー領域のメモリもいくつかのセグメントに分割して使用します．メモリの管理とは，メモリ領域をどのように分割するかを決めることです．いくつのセグメントに分割し，プログラムのどの部分をどのセグメントに割り当てるかを考えなければなりません．

たとえば**図4-3**のように分割することにしましょう．各セグメントをそれぞれC，D，Sという名前で表すことにします．各セグメントにプログラムのどの部分を割り当てるかは，ここではまだ気にしないことにします．

図4-3に示すようにプログラムで使用するメモリ領域，つまりC，D，Sの3つのセグメント以外は空き領域となります．3章で解説したCOMモデルのプログラムは1つのセグメントだけを使用します．残ったメモリ領域はすべて未使用の空き領域になるわけです．

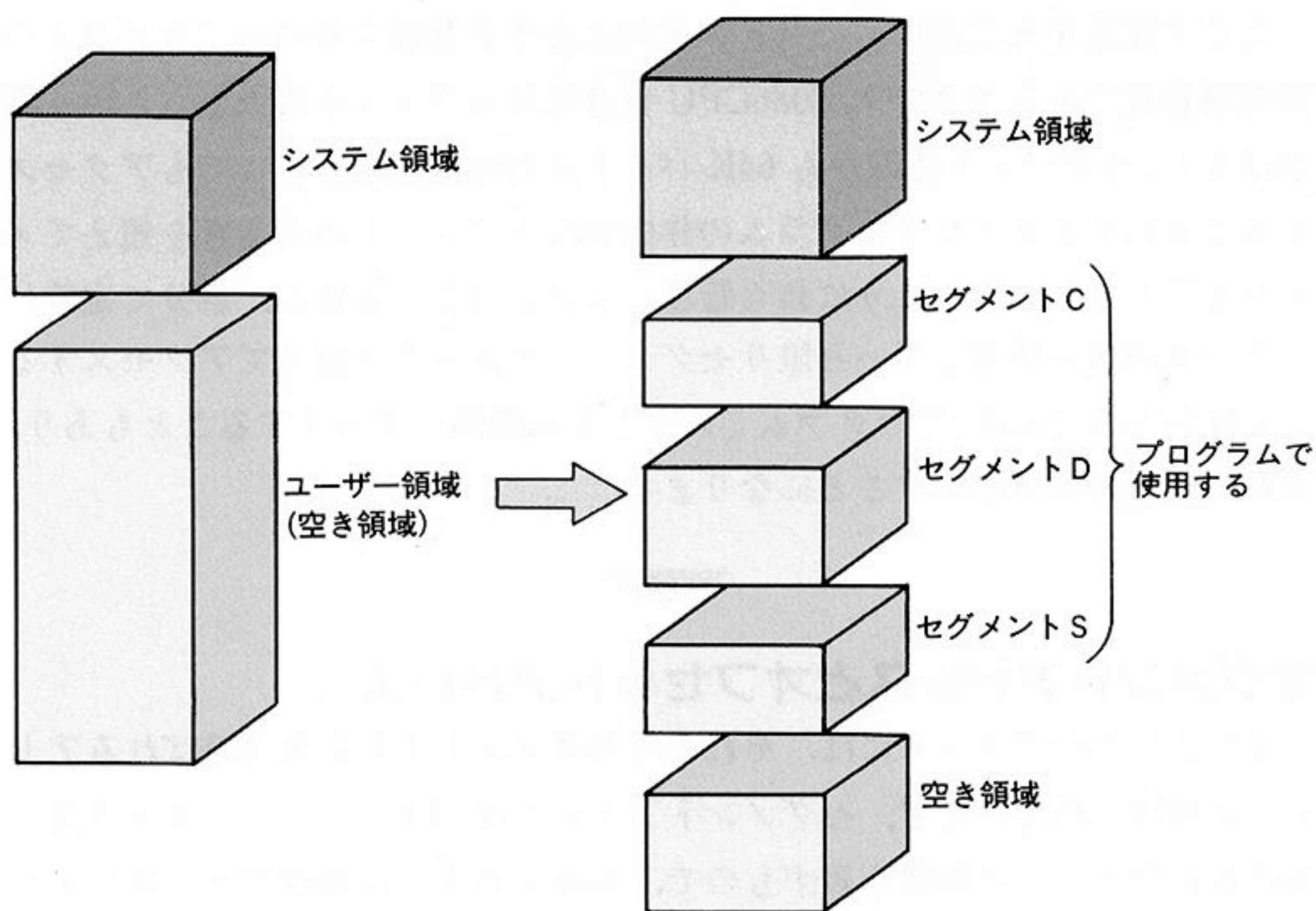



図 4-3　ユーザー領域をセグメントに分割

セグメントの大きさ

各セグメントの大きさは，必ず64Kバイト以内です．64Kバイト「以内」という表現に注目しましょう．図4-3に示したように，セグメントの大きさは一定であるとは限りません．セグメントの大きさは必要に応じて決めることができるので，1Kバイトのセグメントもあれば64Kバイトのセグメントもあるのです．

実際のセグメントの大きさは，プログラムをセグメントに分割した際に各セグメントに置かれたマシン語コードやデータの量によって自動的に決まります．アセンブル&リンクの結果，1つのセグメントの大きさが64Kバイトを超えていると，エラーが発生し実行ファイルを生成することができません．このときはそのセグメントをさらにいくつかのセグメントに分割することになります．

ここで注意することは，セグメントの大きさを管理するのはプログラムの作成者自身であることです．8086CPU 自身にはセグメントの大きさという概念はなく，セグメント先頭から 64K バイト以内の範囲ならどこでもアクセスすることができます*．プログラムの作成者はセグメントの大きさを超えてメモリをアクセスしないように自ら管理しなければなりません．自分で定義したデータ領域を利用している限りセグメントの大きさを超えてアクセスすることはありませんが，プログラムミスなどから偶然アクセスすることもあり，思わぬ結果を引き起こすことになります．

セグメントアドレスとオフセットアドレス

1つ1つのセグメントには，それぞれ**セグメントアドレス**と呼ばれるアドレスが付けられています．セグメントアドレスは 1M バイトのメモリ空間におけるセグメントの位置を表すもので，後述するように**物理アドレス**にそのまま対応します．しかしここでは，図 4-4 のように「セグメントには番号が付いている」とだけ考えておいてください．16 ビット，つまり 4 桁の 16 進数で表される番号がついているのです．

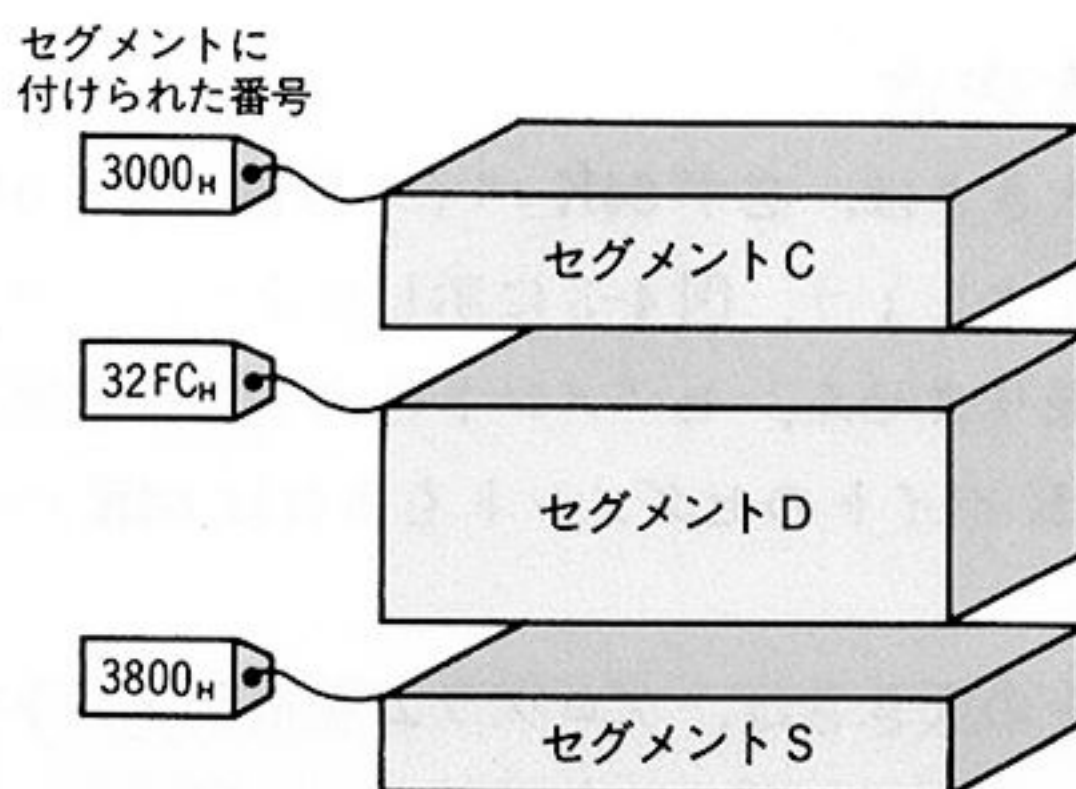



図 4-4　セグメントに付けられた番号（セグメントアドレス）

---

* 80286CPU のプロテクトモードでは，セグメントの大きさは CPU 自身の機能によって管理される．くわしくは APPENDIX 参照．

1つ1つのセグメントは，それ自体独立したメモリ空間であると考えることができます．したがって，**図4-5**のようにセグメントのなかにおけるアドレスを考えます．これが**オフセットアドレス**と呼ばれるアドレスです．

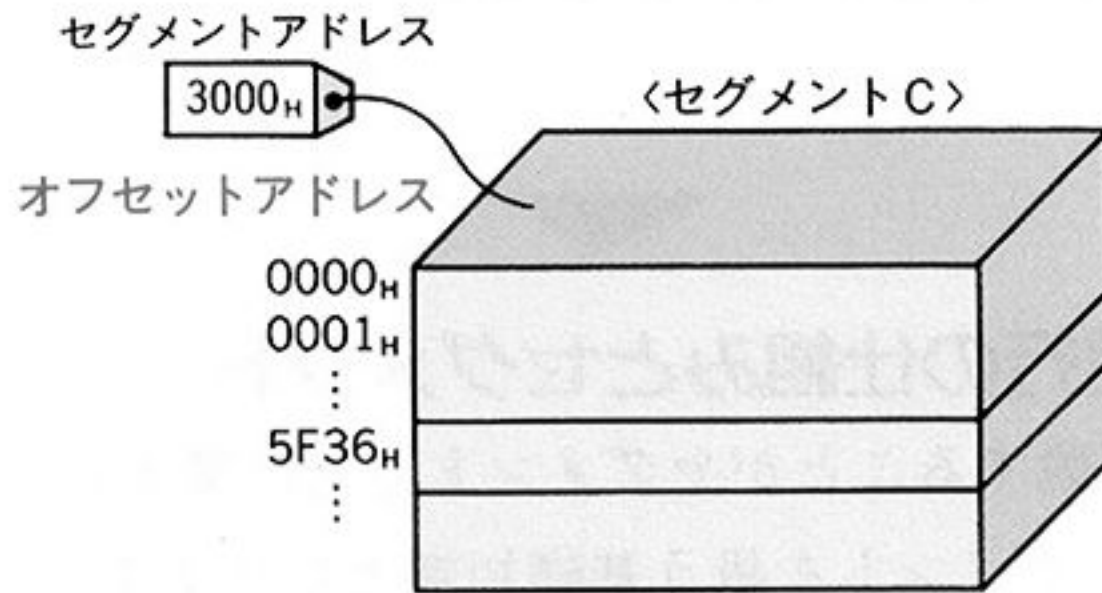



図4-5 セグメント内のアドレス（オフセットアドレス）

オフセットアドレスは，＜0000_H＞から＜セグメントの大きさ－1＞までの数で表します．最大の大きさである64Kバイトのセグメントでは，$FFFF_H$までとなります．これは16ビットで表せる最大の数であり，セグメント内のオフセットアドレスは16ビットのレジスタ1本ですべて表せることを意味します．

なお，オフセットアドレスのことを単にアドレスという場合がありますので注意してください．

4.2

セグメント方式の仕組み

プログラムの実行の仕組みとセグメント

さて，これから解説することがセグメント方式に関するもっとも重要なところで，MASM でセグメントを扱う基礎知識となります．

CPU の動作は，メモリからマシン語命令を読み込み，それを解釈して実行するという 3 つのステップの繰り返しです．命令のなかにはメモリからデータを読み出す命令や，メモリにデータを書き込む命令のように，メモリとのやりとりを伴うものがあります．8086CPU ではこのようなプログラム実行のステップのなかで，「マシン語命令の読み込み」，命令を実行することによる「データの読み書き」，「スタックの操作」という 3 種類のメモリアクセスをそれぞれ別々のセグメントに対して行います．

たとえば，図 4-6 ①のように 3 つのセグメントがそれぞれ割り当てられているとします．すると，CPU は「セグメント C」から次に実行すべき命令を読み出します(図 4-6 ②)．その命令が「MOV AX, [0200H]」であったとしましょう (図 4-6 ③)．これはアドレス 0200_H のメモリのデータを AX レジスタに転送しろという命令です．CPU はこの命令を解釈して次のように実行します．「セグメント D」のアドレス 0200_H の内容を読み出し，AX レジスタに転送するのです(図 4-6 ④)．そして，次の命令を再び「セグメント C」から読み出し，解釈，実行します．PUSH，POP や CALL，RET などスタックを操作する命令では，図 4-6 ⑤⑥のように「セグメント S」とデータをやりとりします．

このように 8086CPU は，メモリとのデータのやりとりを CPU の動作の種類に応じて 3 種類のセグメントと独立に行います．各セグメントはその役割から，**コードセグメント**，**データセグメント**，**スタックセグメント**と呼ばれます．

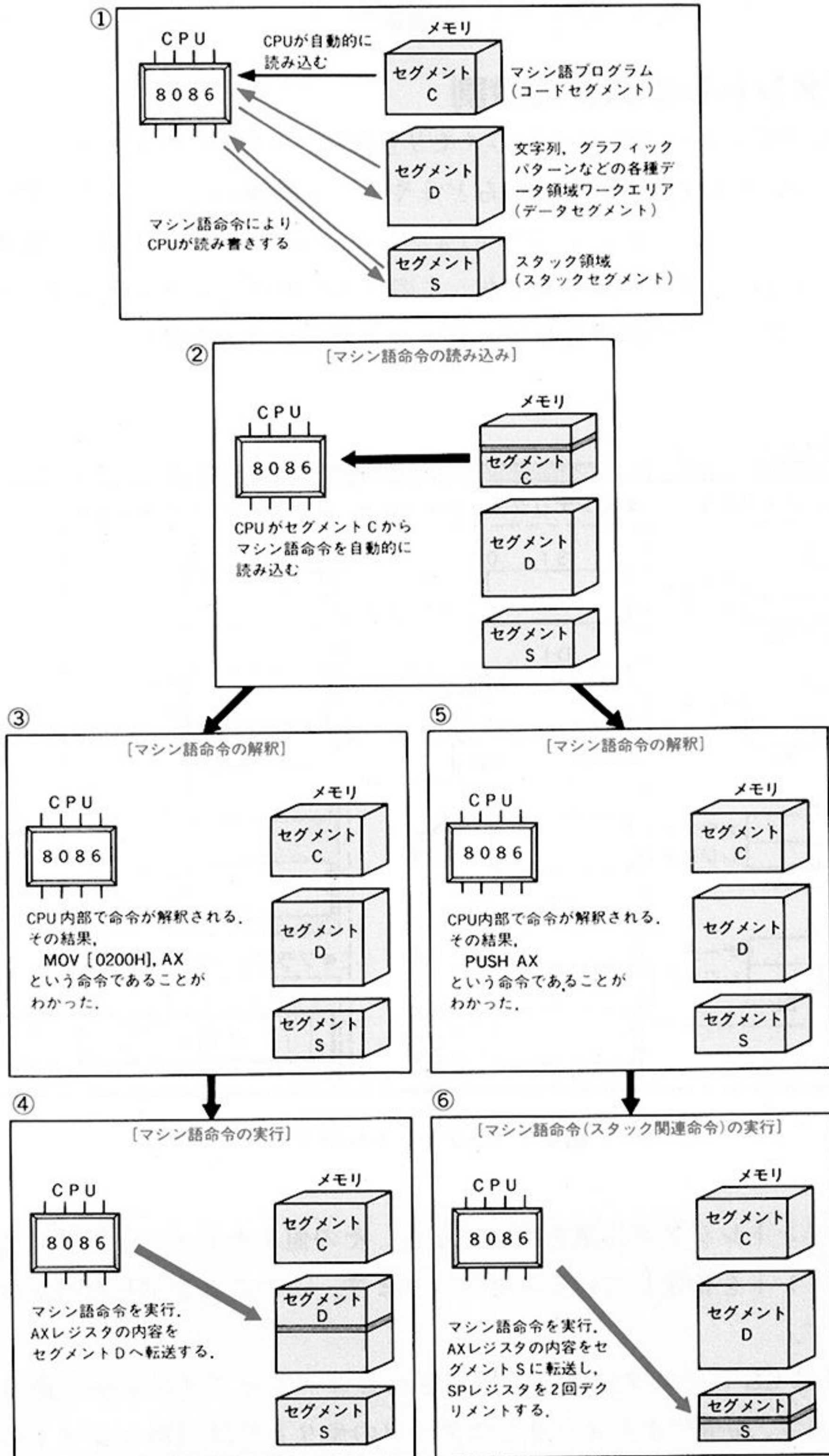



図4-6　3種類のセグメントの役割

セグメントレジスタの役割

8086CPU の持つ 1M バイトのメモリ空間は，いくつものセグメントに区切ることができます．そのなかからどうやって 3 種類のセグメントを指定するのでしょうか．この答えは CPU のなかにあります．8086CPU には**図 4-7** に示すようなレジスタがありますが，このうちの**セグメントレジスタ**と呼ばれるレジスタが，その名のとおりセグメントアドレスを決定します．

図 4-7　セグメントレジスタ

セグメントレジスタに値を設定すると，その値をセグメントアドレスとするセグメントを指定していることになります．このことを示したのが次の図 4-8 です．

CPU は CS レジスタによって指定されるコードセグメントから命令を読み込みます．命令によるメモリとのデータのやりとりは，DS レジスタの指定するデータセグメントと行います．同様に SS レジスタの指定するセグメン

図4-8 セグメントレジスタとセグメントの関係

トがスタックセグメントです．

CPUの行う3種類のメモリとのやりとりは，このようにCS, DS, SSレジスタが指定するセグメントに対して行われることを示したのが次の図4-9です．なお，セグメントレジスタにはもう1つESレジスタがありますが，このESレジスタの役割についてはもう少しあとで解説します．

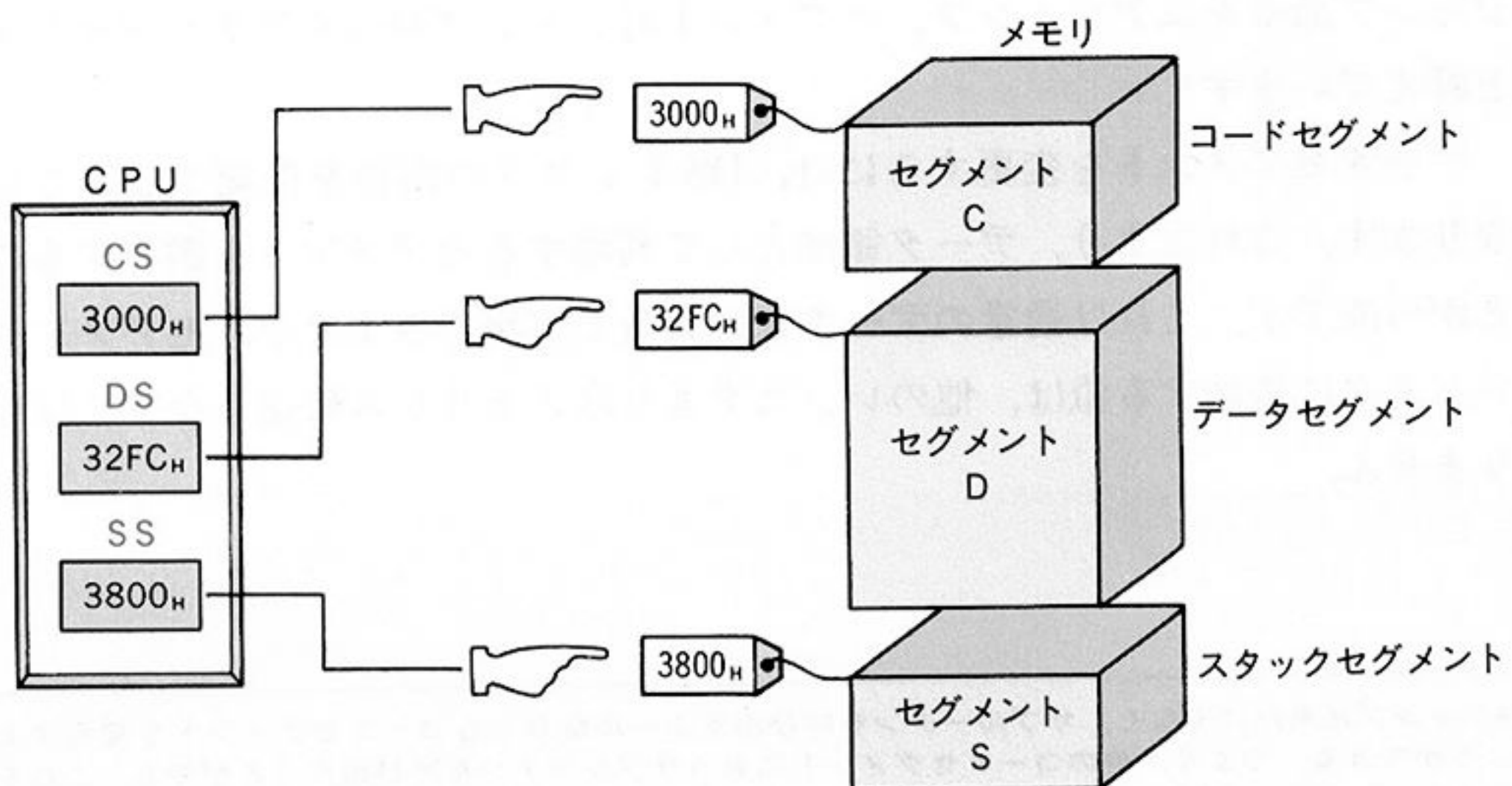



図4-9 CS, DS, SSレジスタの役割

セグメントの選択方法

8086CPUのセグメント方式では，プログラムに必要なメモリをコード，データ，スタックの3種類に分けるだけでなく，それぞれをさらにいくつかのセグメントに分割することができます．つまり，複数のコードセグメントやデータセグメントを利用することができるのです．ただし，同時には3種類のセグメントをそれぞれ1つずつしか利用できませんから，セグメントレジスタの内容を変更することにより利用するセグメントを切り替えていくことになります．ここではその方法を解説します．

コードセグメントを変更するということは，CSレジスタの内容を変更することです．しかし，それはマシン語プログラムを読み込むアドレスを変更することになるので，IPレジスタと同様にデータ転送命令ではできません．これを実現するマシン語命令はジャンプ命令で，それも「セグメント外ジャンプ命令」と呼ばれる命令です．単なるジャンプ命令では，IPレジスタにジャンプ先のオフセットアドレスがセットされますが，セグメント外ジャンプ命令ではさらにCSレジスタにもジャンプ先のセグメントアドレスがセットされます．つまり，セグメント外ジャンプ命令によってプログラムの実行位置が他のセグメントにジャンプするわけです．このため通常の(セグメント内)ジャンプ命令を**ニアジャンプ**，セグメント外ジャンプ命令を**ファージャンプ**と呼んでいます*．

データセグメントを変更するには，DSレジスタの内容を変更することになります．これにより，データ領域として利用するセグメントを選択することが可能です．これは通常のデータ転送命令で実現できますが，セグメントレジスタに格納する値は，他のレジスタまたはメモリから転送しなければなりません．

*ジャンプ命令だけでなく，サブルーチンを呼び出すコール命令でもコードセグメントを変更することができる．つまり，他のコードセグメントにあるサブルーチンを呼び出すことができ，これを「ファーコール」と呼ぶ．他のコードセグメントから呼び出されるサブルーチンのリターン命令は当然ながらIPレジスタおよびCSレジスタ双方の内容をスタックから復帰する「ファーリターン」命令でなければならない．くわしくは5章で解説する．

最後にスタックセグメントについてですが，通常のプログラムではスタックセグメントを変更することはありません．プログラムの最初でSSレジスタにスタックセグメントのアドレスをセットするだけです．しかもプログラムのロード時にMS-DOSシステムがアドレスをセットしてくれるので，結局スタックセグメントに関する操作は必要な領域を確保しておくことだけです．

MS-DOSの実行型ファイルには，ファイル拡張子によって区別される「COMモデル」と「EXEモデル」の2種類がありますが，両者の違いは利用できるセグメントの数にあります．図4-10に示すようにCOMモデルのプログラムでは1つのセグメントしか利用できないのに対し*，EXEモデルでは複数のセグメントを利用できます．

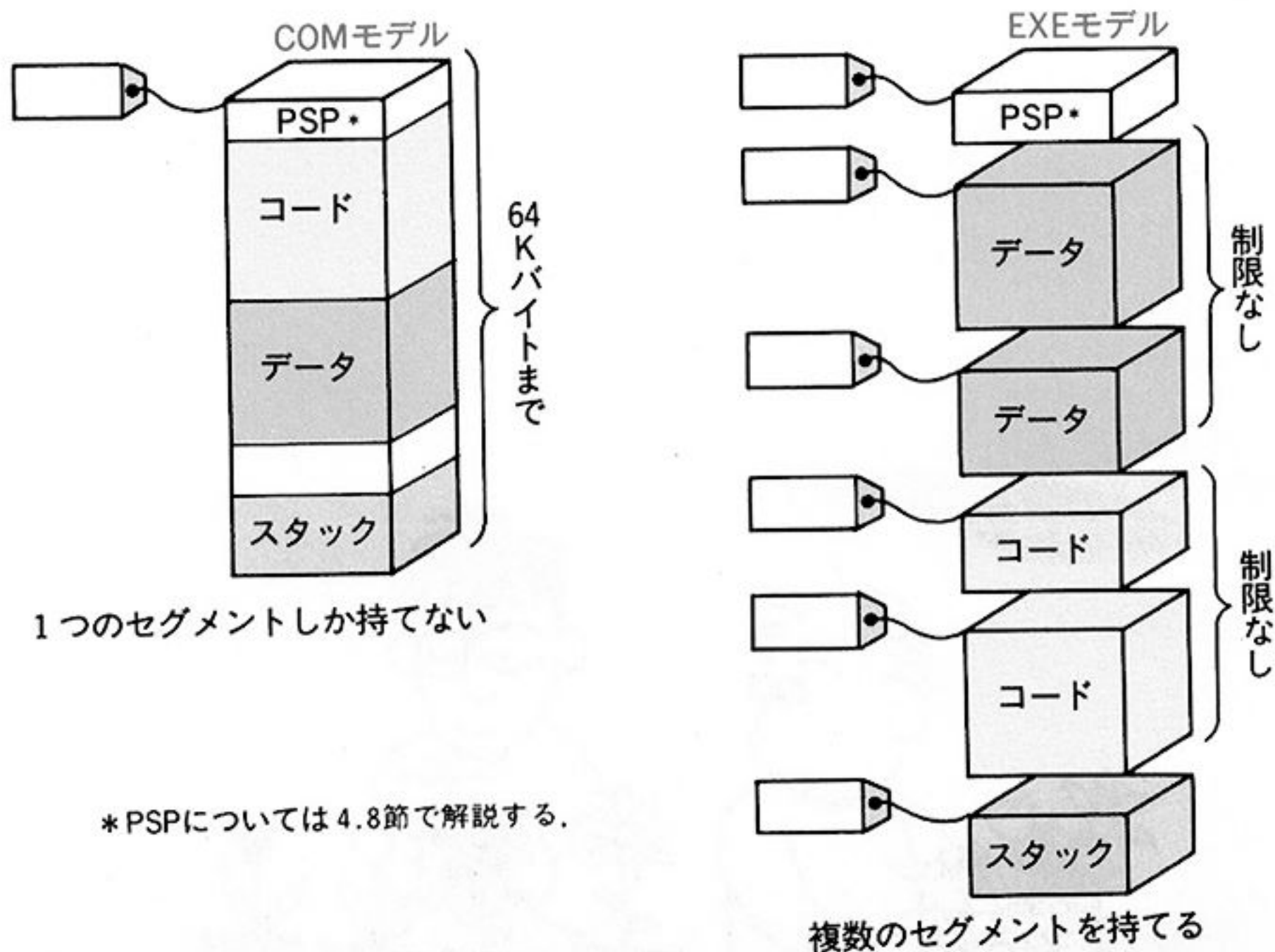



図4-10　COMモデルとEXEモデル

＊COMモデルのプログラムがセグメントを1つしか持てないわけではなく，実行ファイルに含まれるセグメントが1つということ．COMモデルでも，実行時に空き領域からセグメントを確保すれば，複数のセグメントを扱うことが可能．

本章の最初で触れたように，複数のセグメントを利用することによって初めて64Kバイトを超えるユーザー領域のメモリをすべて利用することができます．そのためにはEXEモデルのプログラムを作成しなければなりません．4.3節以降の節では，このEXEモデルのプログラムを作成するために必要な知識を解説していきます．

なお，すべてのメモリが利用できるからといってEXEモデルの方が優れているとは必ずしもいいきれません．くわしくは本章の後半で解説しますが，複数のセグメントを扱うことによるデメリットも存在するからです．逆にCOMモデルにもそれなりにいろいろなメリットがあります．このように，作成するプログラムの種類に応じて使いわけるために2種類の実行ファイル形式が用意されているわけです．

物理アドレスとセグメントアドレス

これまでセグメントアドレスはセグメントに付けられた番号として扱ってきました．その理由は，MASMがセグメントアドレスを概念的に扱う機能を持っており，セグメントアドレスと物理アドレスの対応を意識する必要はないからです．

しかし，プログラムのデバッグにおいては知っていると便利ですし，8086CPUのセグメント方式の弱点を把握する上でも知っておいて損はありません．また，物理アドレスの固定されたVRAM領域などをアクセスする場合には，両者を対応させる必要があります．

ここでは物理アドレスとセグメントアドレスの関係をわかりやすく解説することにします．すでにわかっている人は飛ばしてもらってもかまいません．

8086CPUの持つ1Mバイトのメモリ空間には，図4-11に示すように**物理アドレス**という通し番号が付いています．

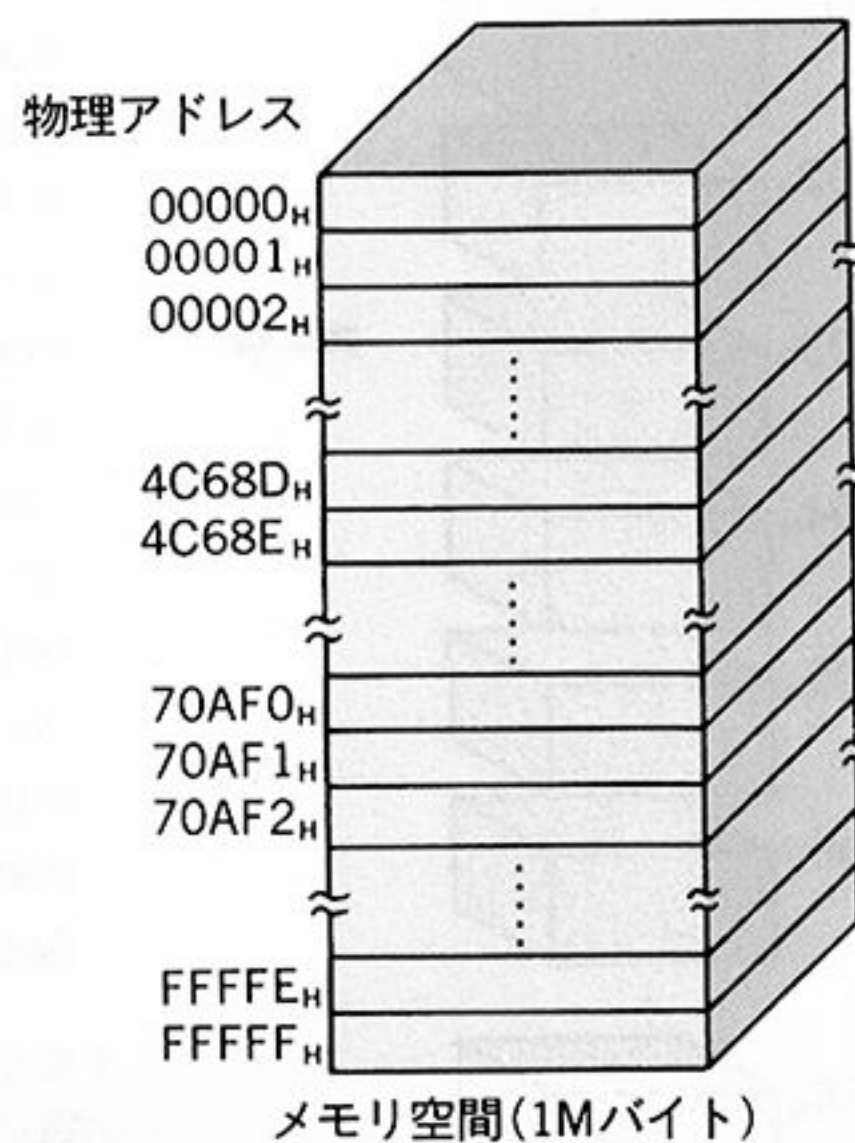



図4-11 物理アドレス

物理アドレスは20ビットの値であり，16ビットのレジスタしか持たない8086CPUでは直接扱うことができません．そこでセグメント方式が登場するのですが，次のように考えると非常にわかりやすくなります．

まず，メモリ空間を図4-12のように16バイトごとの細かい領域に分割することを考えます．そして，各領域に0から始まる番号を付けます．するとその番号は$FFFF_H$までとなり，ちょうど16ビットの範囲に納まります．この領域のことを「パラグラフ」と呼びます*．

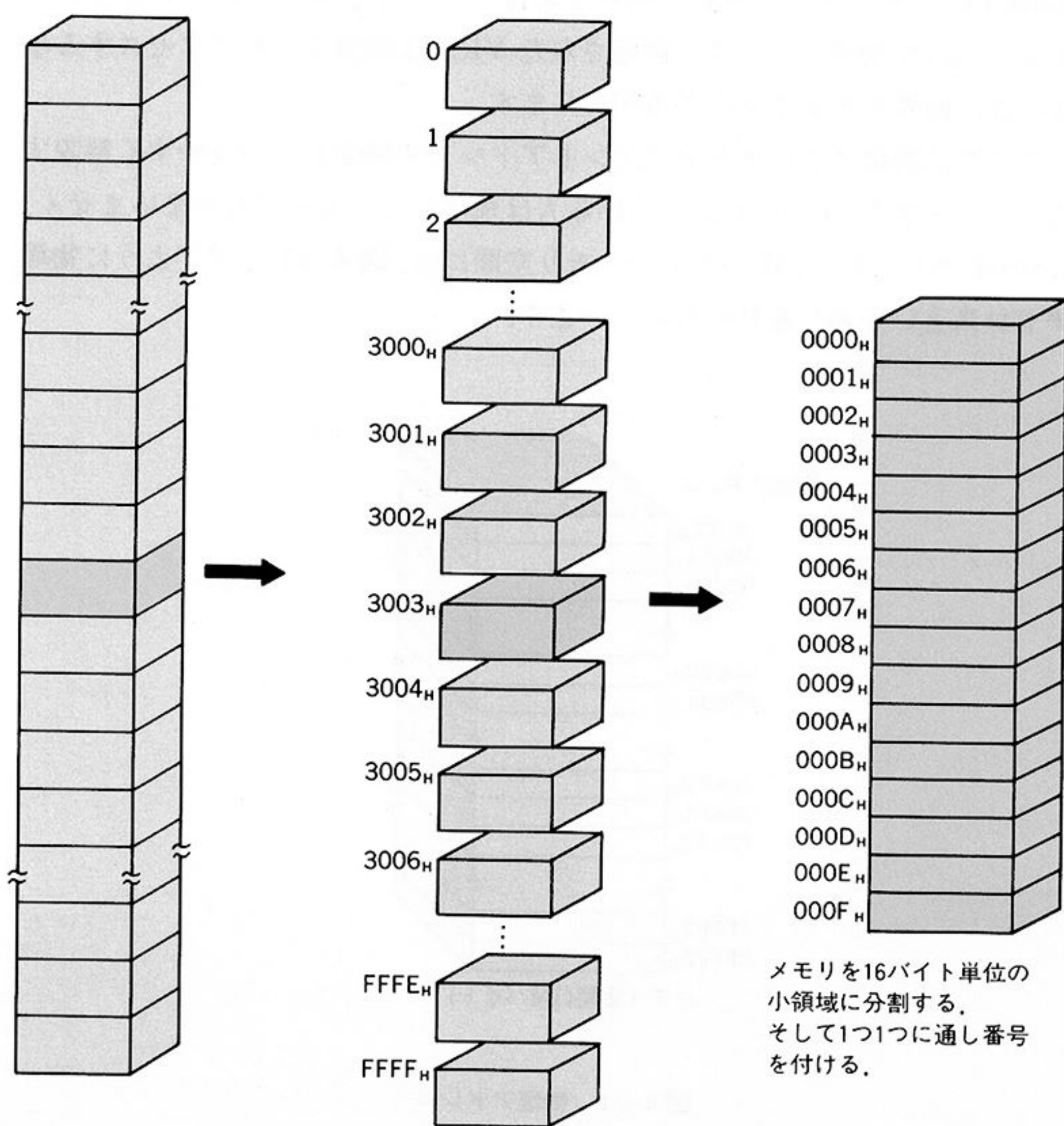



図4-12　物理アドレスをパラグラフに分割

次に，この小領域をグループ分けすることを考えましょう．図4-13のように連続した小領域をまとめていくつかのグループを作るのです．

各グループはそれぞれが1つの独立したメモリ空間であると考えることができます．これがセグメントです．そして，各グループの先頭の小領域の番号がそのセグメントのセグメントアドレスになります．

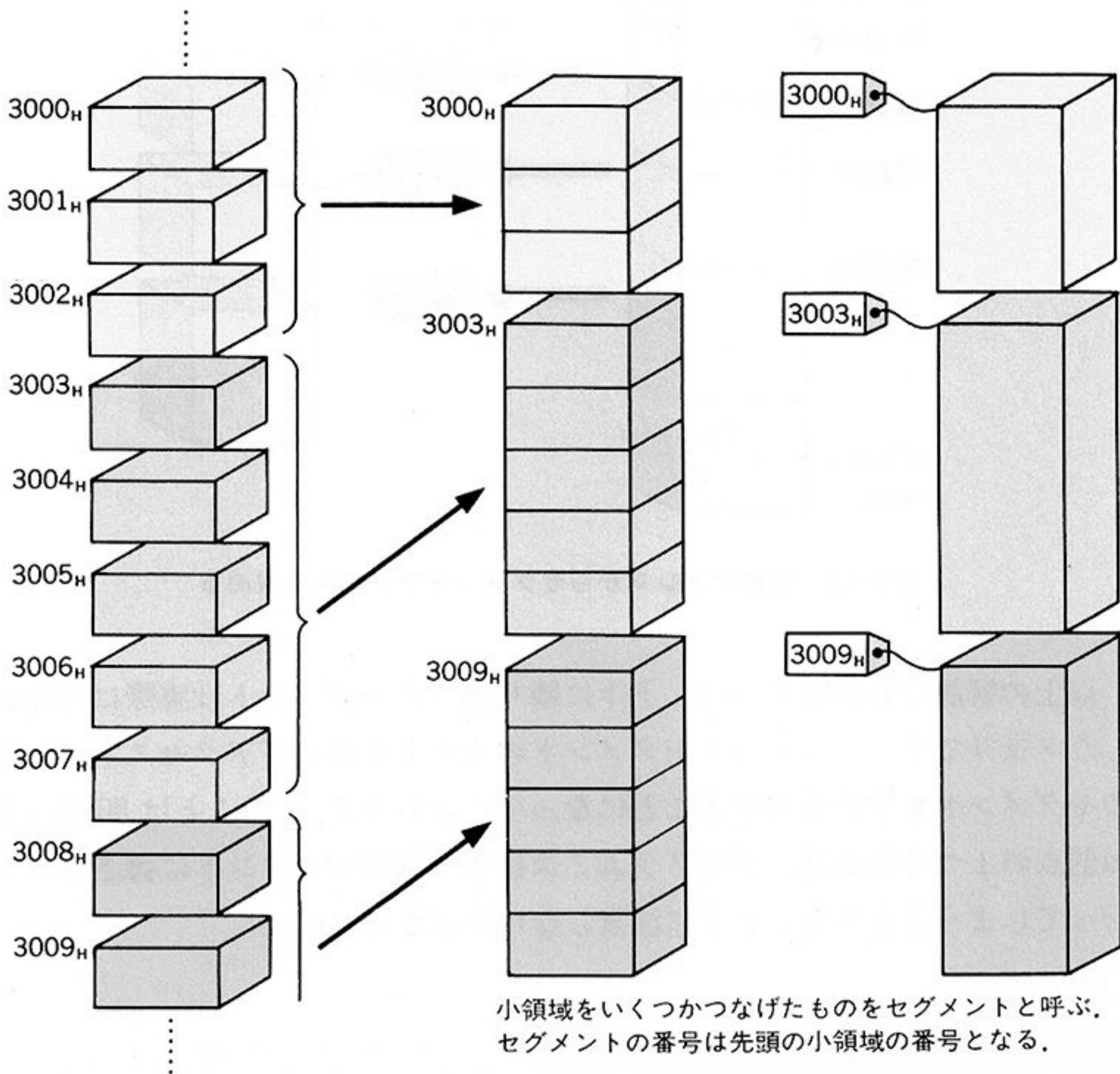



図4-13　パラグラフとセグメント

＊メモリのサイズをバイト数ではなく，パラグラフ数で表すことがある．たとえば，6.2章で紹介するプログラムで利用している常駐終了のファンクションコールでは，プログラムの大きさをパラグラフ数で指定する．

小領域の大きさはそれぞれ16バイトですから，セグメントアドレスを16倍した値が物理アドレスになります．16進数で言えば，1桁ずらして0を付けることになります．逆に，16進数の物理アドレスから下1桁を落とすだけで，セグメントアドレスに変換したことになります(図4-14)．

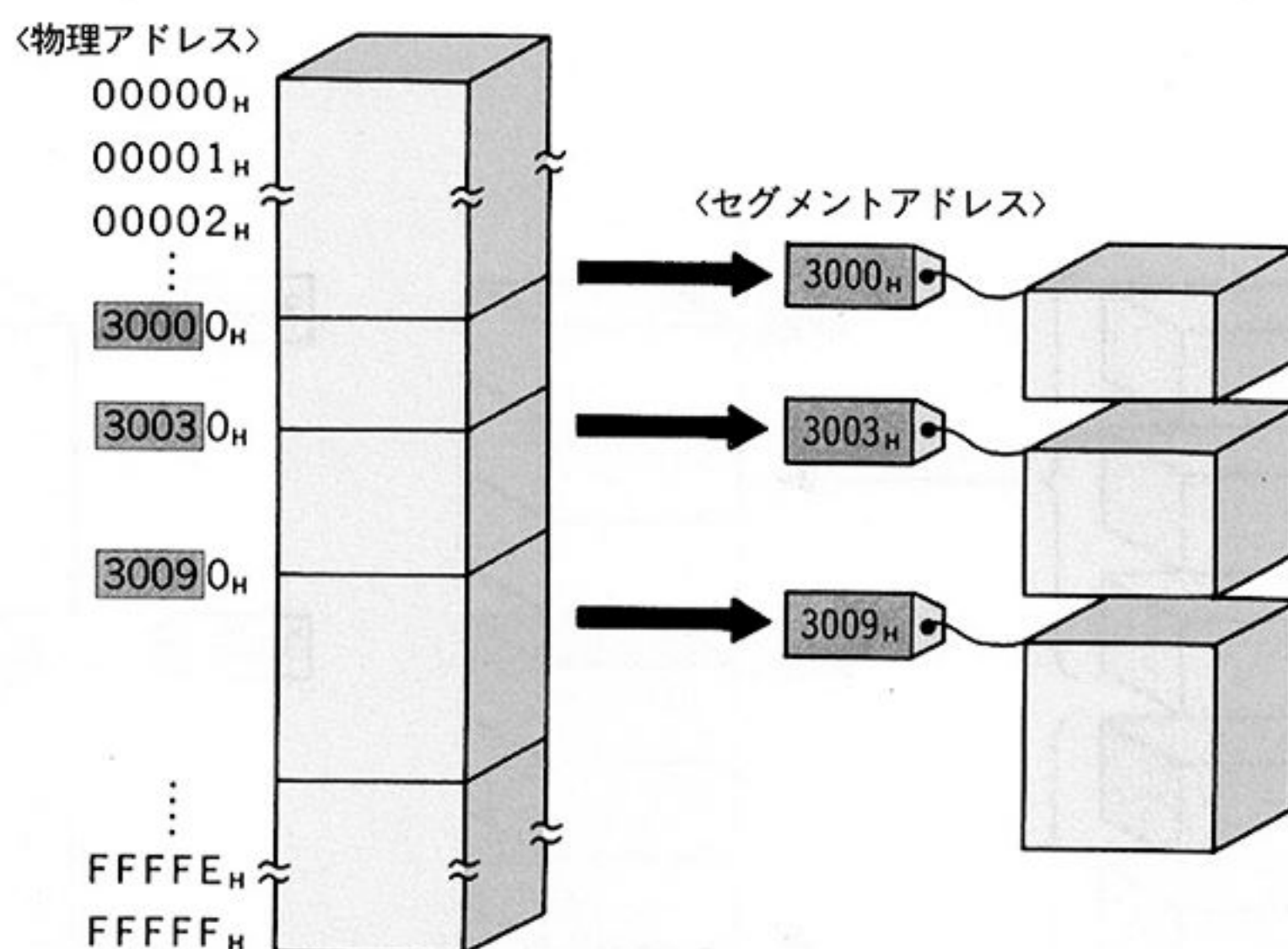



図4-14　物理アドレスからセグメントアドレスを求める

以上の解説でわかるように，上下に隣り合ったセグメントは実際には連続したメモリです．したがってセグメントの大きさを超えてアクセスすると隣のセグメントをアクセスすることになってしまいます．このことは8086CPUの弱点の1つでもあり，プログラムミスにより隣のセグメントに書き込みを行ってしまうことのないように注意しなければなりません*.

*すでに114ページでも解説している．

4.3
SEGMENT 擬似命令

本節以降では，MASM はどのようにセグメントを扱うのかを解説していきます．MASM は 8086CPU のセグメント方式をサポートする強力な機能を持っており，特にセグメントアドレスを概念的に表す機能には多くのメリットがあります．プログラム中で定義したセグメントがセグメントアドレスとして具体的にどんな値を持つのか，その物理アドレスはどんな値かといったことを意識する必要はないからです．しかも，セグメントアドレスを概念的に扱うことにより，80286CPU のプロテクトモードにおいてもほぼ同じような考え方でプログラムを作成することができます．

MASM におけるセグメントの扱い方は決して難しいものではありませんから，よく読んで理解してください．

セグメントの定義

［書式］　セグメント名　SEGMENT
　　　　　　　　　　　　⋮
　　　　　セグメント名　ENDS

・セグメント名は｛アルファベット，@，$，＿，?，数字｝からなる文字列で，数字で始まることはできない．

MASM ではセグメントをセグメントアドレスではなくセグメントの名前で表します．セグメントの名前は自分の好きな名前を付けることができ，いわばセグメントに付けるラベルのようなものです(図 4-15)．前節ではセグメントにはセグメントアドレスという「番号」を付けて区別することを解説しましたが，MASM では番号ではなく「名前」を付けるのです．

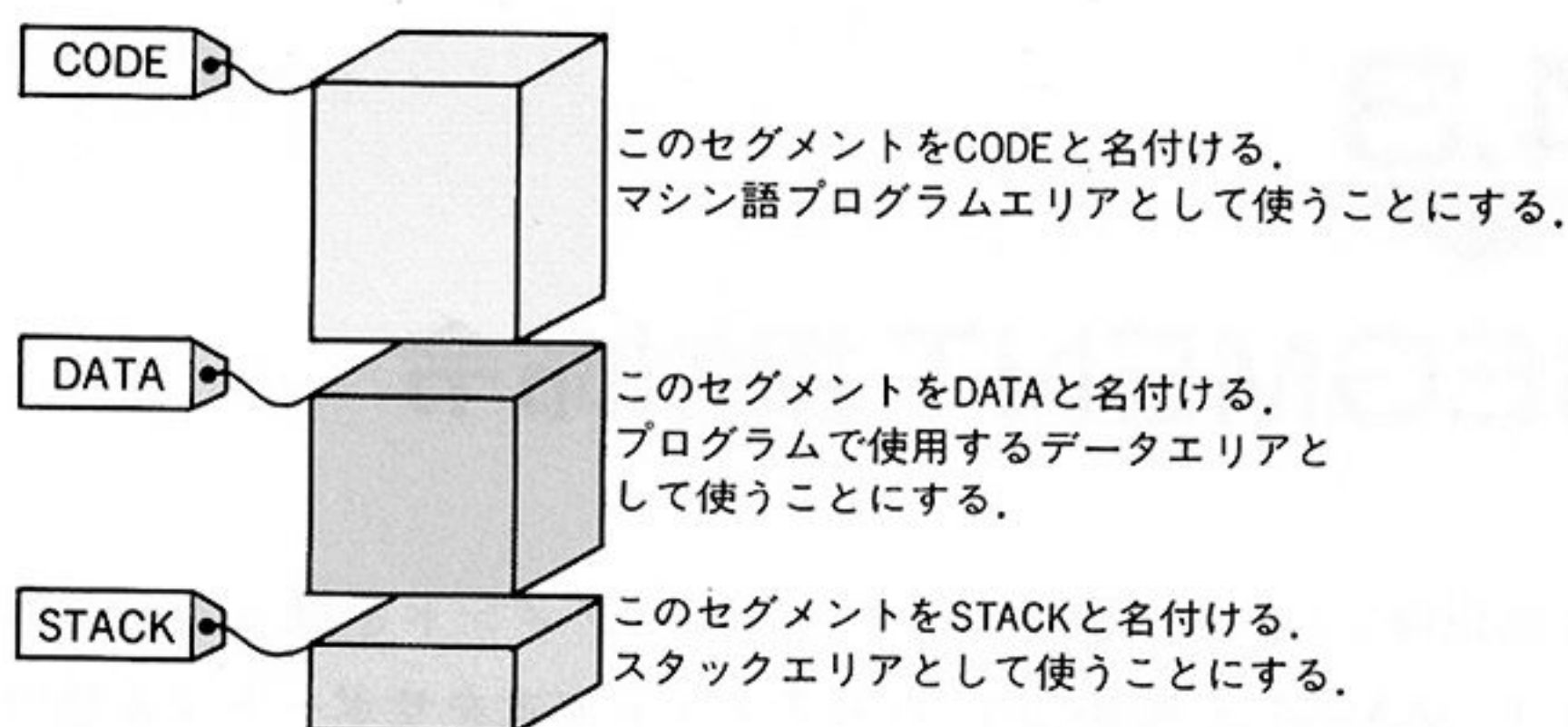
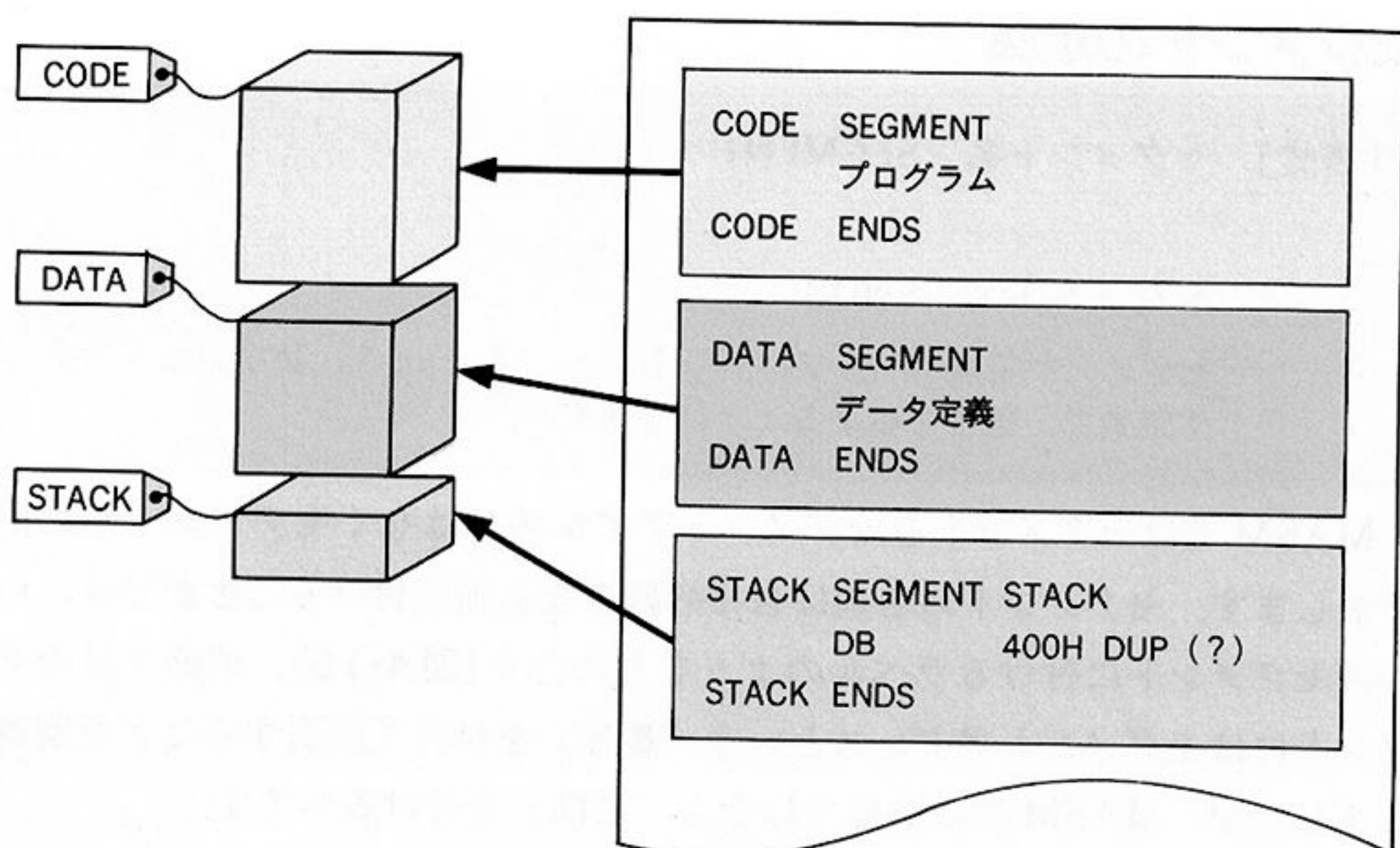



図 4-15　セグメントに名前を付ける

セグメントに関する擬似命令の第1はSEGMENT擬似命令です．SEGMENT擬似命令の役割を図4-16に示します．SEGMENT擬似命令の役割の1つは，セグメントに名前を付けることです．セグメントの名前はラベルと同じように好きな名前を付けることができます．ここでは図に示すように，コードセグメントを「CODE」，データセグメントを「DATA」，スタックセグメントを「STACK」という名前にしました．

もう1つの役割は同時にセグメントの範囲を指定することです．SEG-

図 4-16　SEGMENT 擬似命令の役割

MENT 擬似命令は必ず ENDS 擬似命令と対で使用し，2つの擬似命令で囲まれた部分に書かれたマシン語命令やデータはそのセグメントのなかに定義したことになります．

3章で解説した，プログラムの中身にあたる部分，つまりアセンブル後にできる実行ファイルに含まれるコードやデータは，すべていずれかのセグメントに属していなければなりません．

セグメントごとに名前を付ける

同じ名前のセグメントをいくつ定義しても，1つの連続したセグメントとして扱われます．つまり，ラベルと違って2重定義エラーにはならず，以前のセグメント定義の続きとして扱われるのです．図4-17のようにソースプ

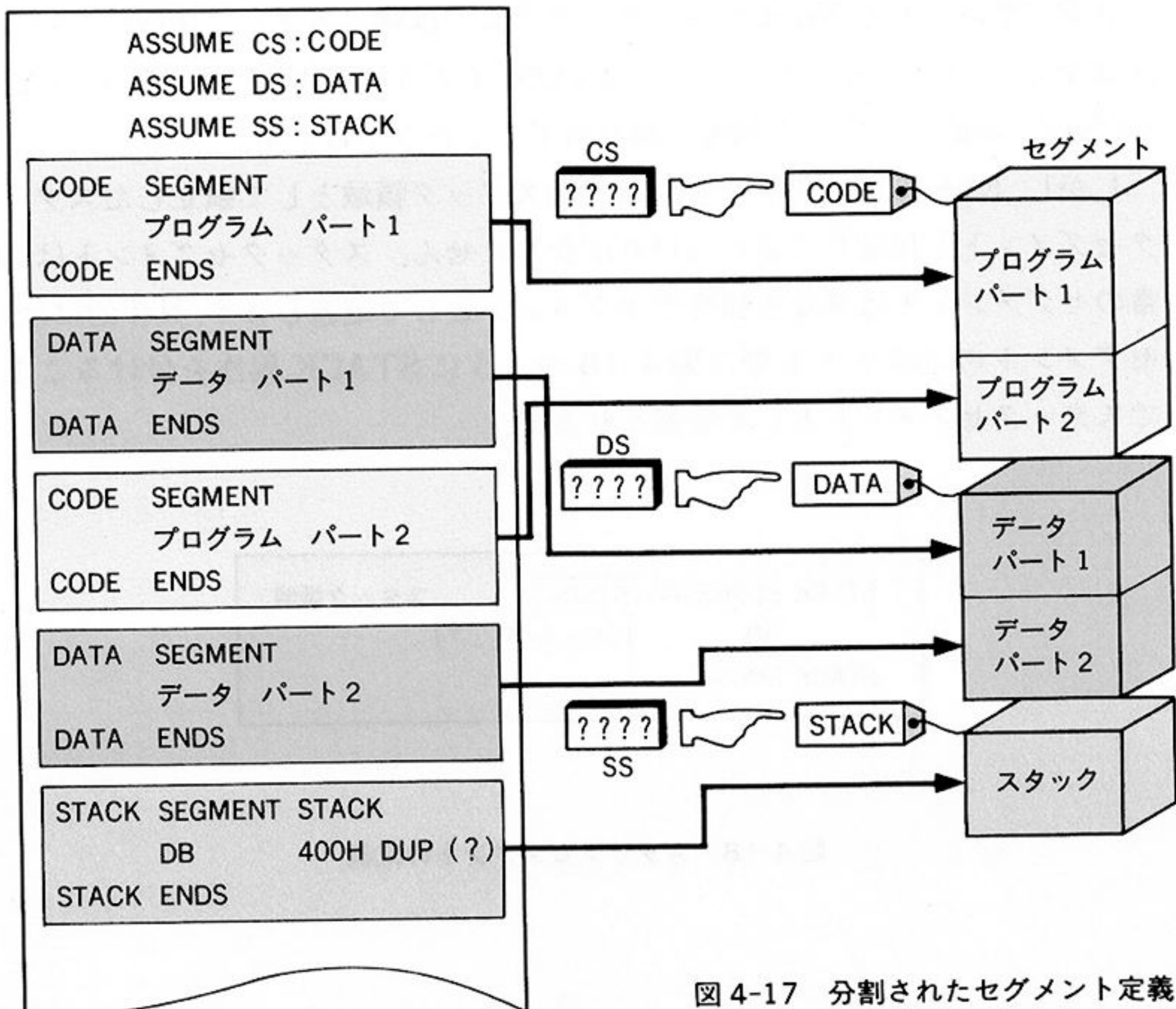



図4-17 分割されたセグメント定義

ログラム中でセグメントの定義を何箇所かに分けて書くと，MASM は同じ名前のセグメントをそれぞれ1つにまとめてしまいます．

したがって，複数のセグメントを定義するためには同じセグメントを分割して定義するのではなく，それぞれに異なる名前を付けます．

スタックセグメントの定義

［書式］ セグメント名 SEGMENT STACK
　　　　　　　　：
　　　　セグメント名 ENDS

・セグメント名は｛アルファベット，@，$，_，?，数字｝からなる文字列で，数字で始まることはできない．

3章で解説した COM モデルのプログラムでは特にスタック領域について注意をはらいませんでした．これは COM モデルのプログラムでは，MS-DOS が自動的にスタック領域を割り当ててくれるからです．

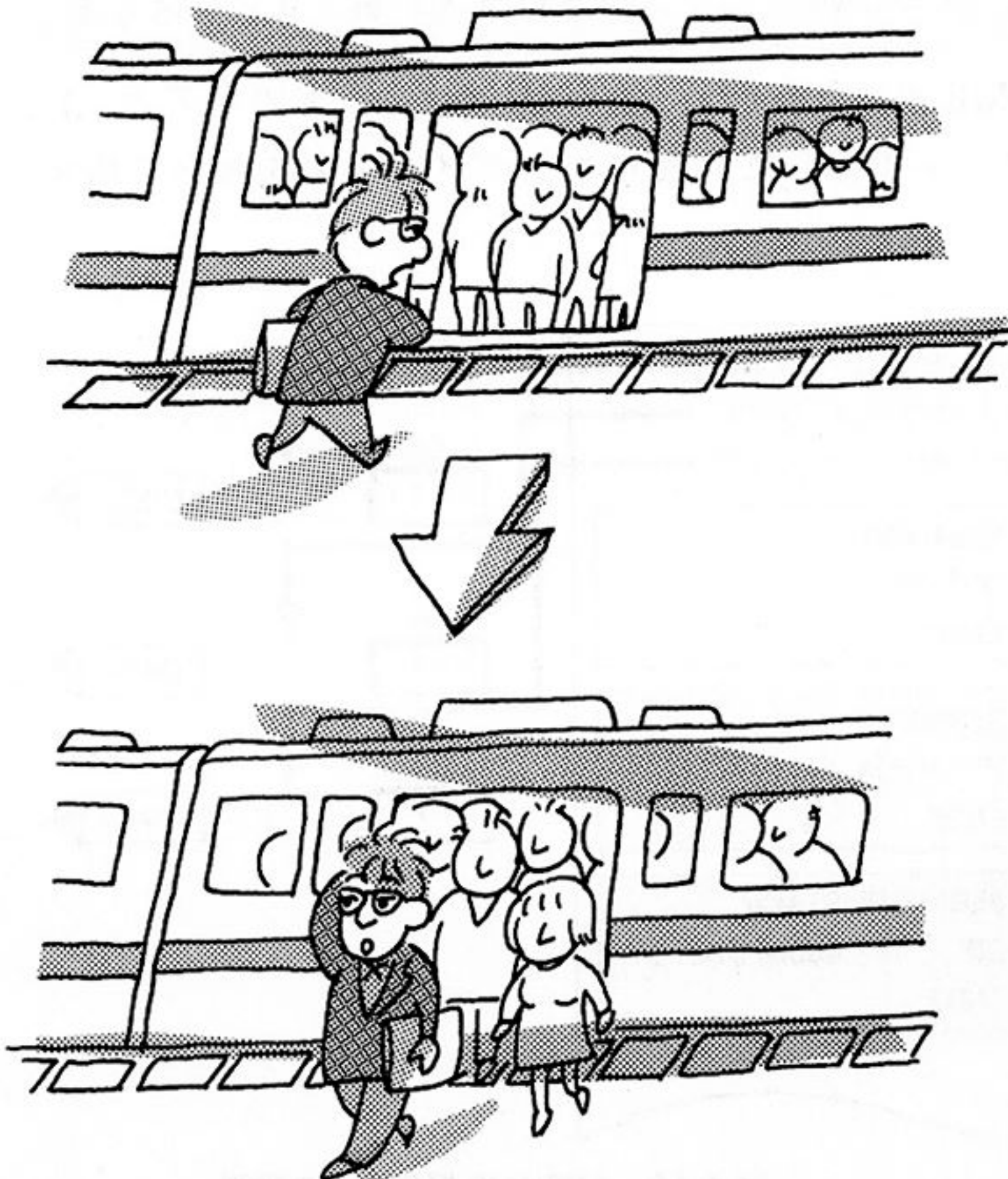
しかし，EXE モデルのプログラムではスタック領域として独立したスタックセグメントを用意しておかなければなりません．スタックセグメントは通常のセグメントとは異なる特殊なセグメントとして定義します．具体的には，セグメントの定義をする際に図 4-18 のように STACK 属性を付けることでスタックセグメントとして定義されます．

```
STACK SEGMENT  STACK          ; スタック領域
      DB       100H DUP (?)
STACK ENDS
```

図 4-18 スタックセグメントの定義

スタックセグメントではDB擬似命令などのデータ領域を確保する擬似命令によって必要なサイズのメモリを確保します．スタックセグメント内に確保したデータ領域の大きさが，そのままスタック領域の大きさとなります．

スタック領域は，CALL命令やPUSH命令によって使用されますが，さらにハードウェア割り込みによっても使われます．したがって，プログラムで使用する大きさよりも余分にメモリを確保しておく必要があります．

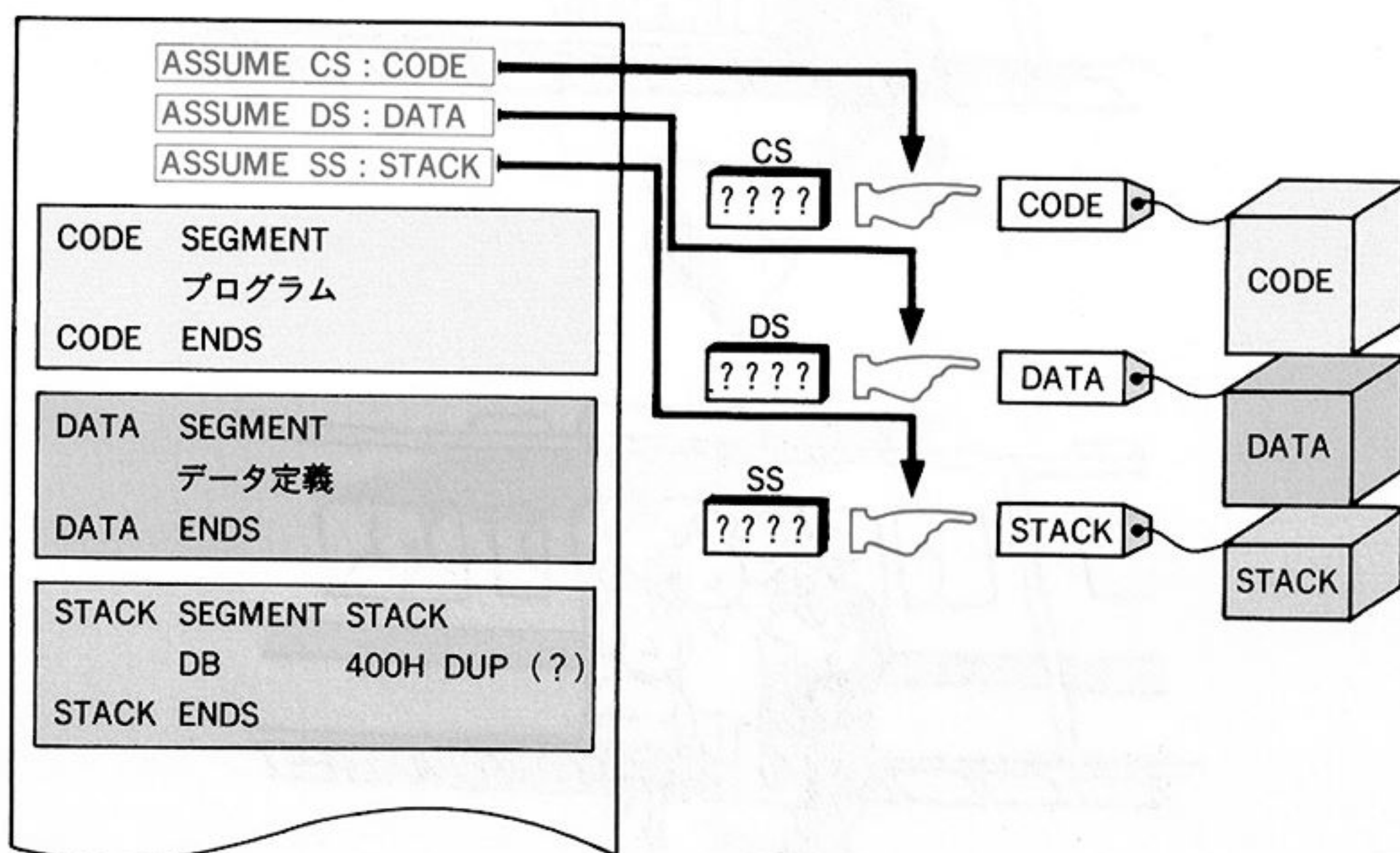
なお前節でも解説したように，スタックセグメントに関してはSTACK属性を付けたセグメントとして領域を確保しておくだけで，それ以外の操作は必要ありません．MS-DOSがプログラムロード時に確保した領域を指すようにSSレジスタやSPレジスタをセットしてくれます．

ASSUME 擬似命令

ASSUME 擬似命令は，4.2 節で解説したセグメントの種類とセグメントレジスタの役割に関する擬似命令です．セグメントレジスタの役割を理解していればASSUME 擬似命令の役割も必ず理解できます．

ASSUME 擬似命令の役割

［書式］ ASSUME　セグメントレジスタ名：セグメント名

ASSUME 擬似命令では，図 4-19 のようにどのセグメントレジスタがどのセグメントを指しているかをセグメント名を使って指定します．AS-

図 4-19　ASSUME 擬似命令の役割

SUME 擬似命令は一種の宣言であり，セグメントレジスタへの値のセットに先立って宣言しなければなりません．

ここで誤解しないように注意すべきことは，ASSUME 擬似命令によって実際にセグメントレジスタが指定したセグメントを指すようになるわけではないことです．ASSUME 擬似命令は，セグメントレジスタとセグメントの対応を MASM に教えることにより，セグメントレジスタが指定したセグメントを指していると仮定(ASSUME)してアセンブルを行うための擬似命令です．ASSUME 擬似命令だけではセグメントレジスタの内容は決定されません(図 4-19)．

データセグメントの選択

セグメント DATA をデータセグメントとして利用するためには，DS レジスタにそのセグメントアドレスをセットしなければなりません．これは図 4-20 に示すように，セグメントアドレスが必要な場所にセグメント名 DATA を書けばよいのです．ニーモニックのなかでセグメント名を使用すると，セグメントアドレスを参照したことになります．

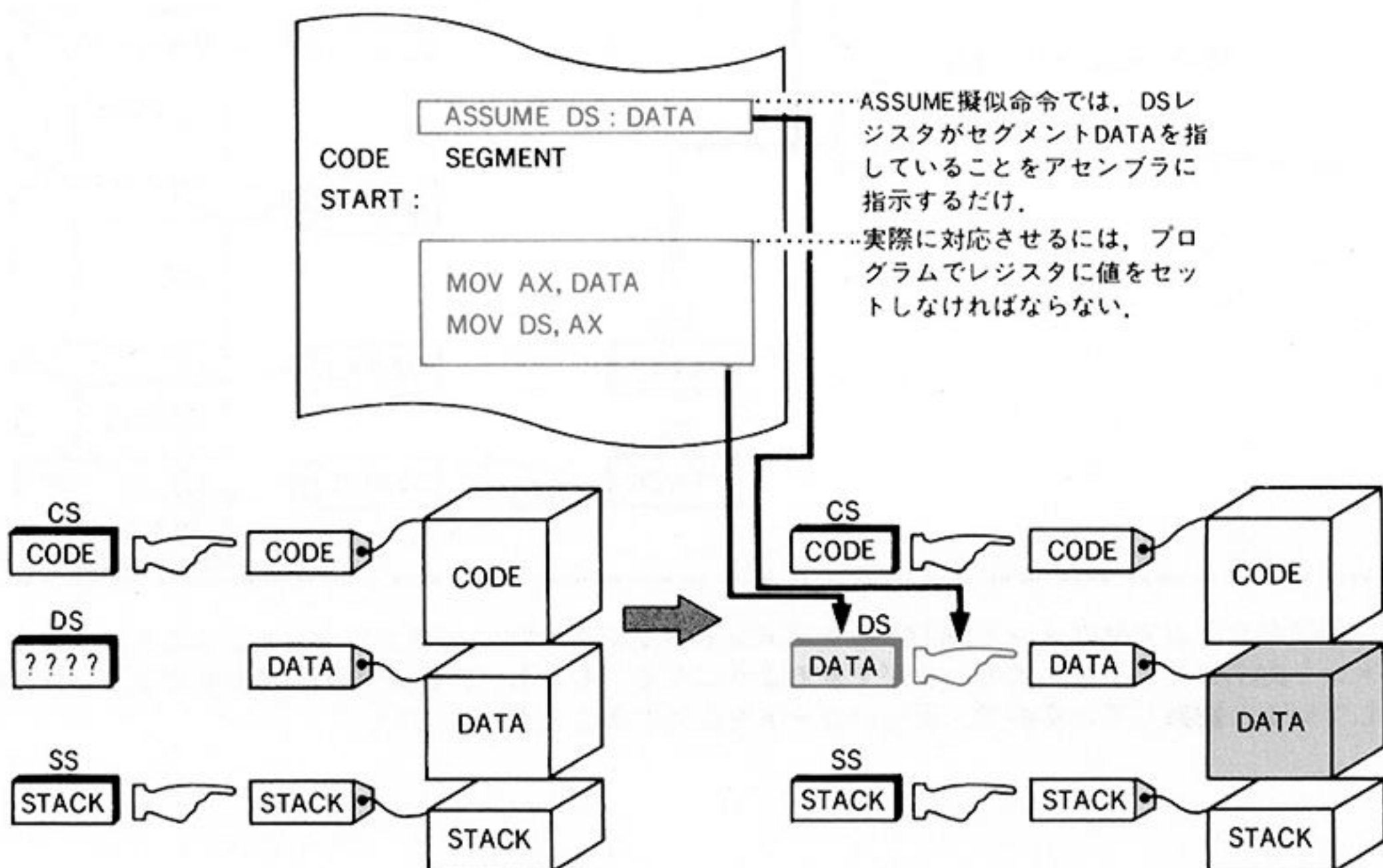



図 4-20 データセグメントの選択

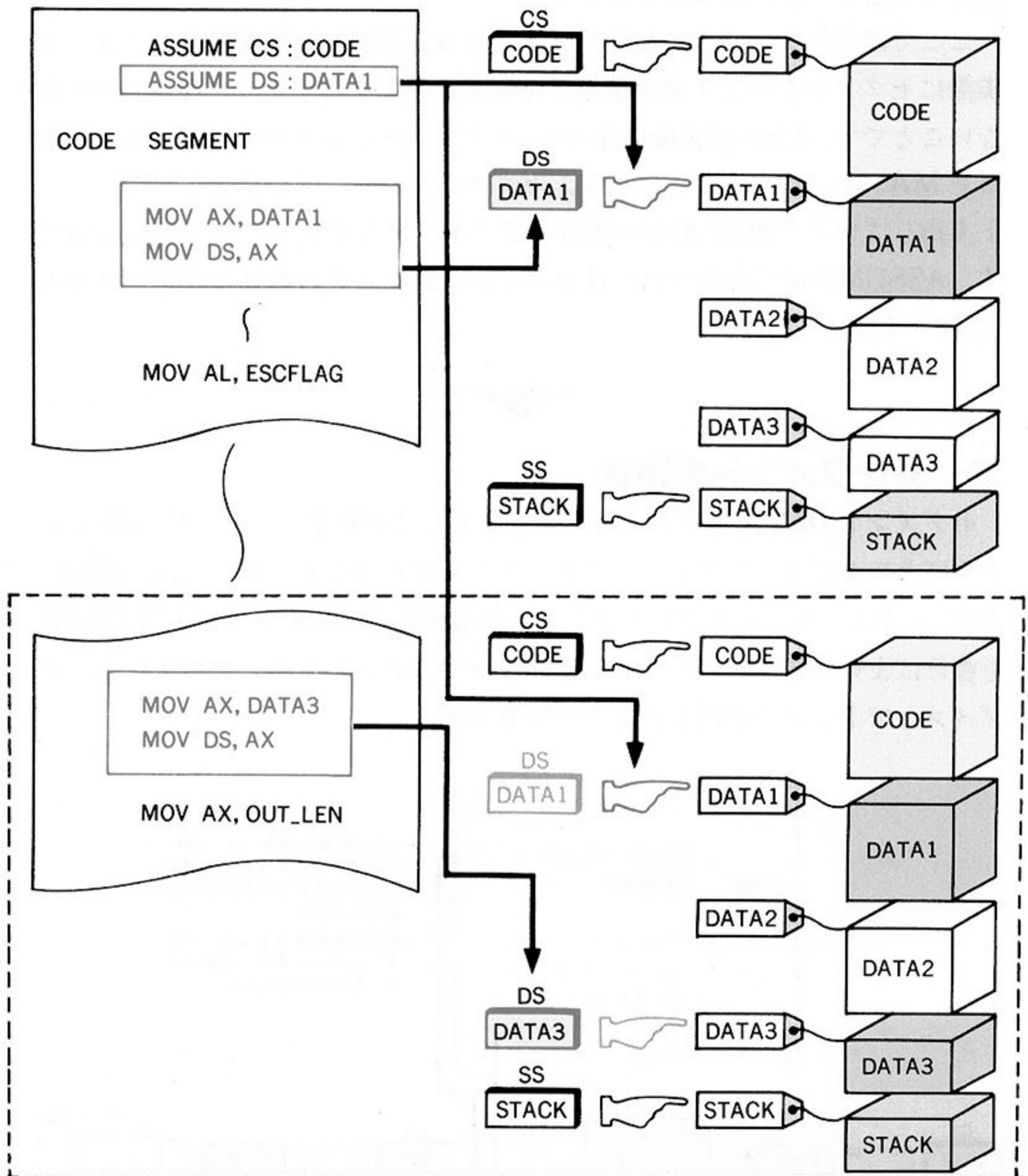
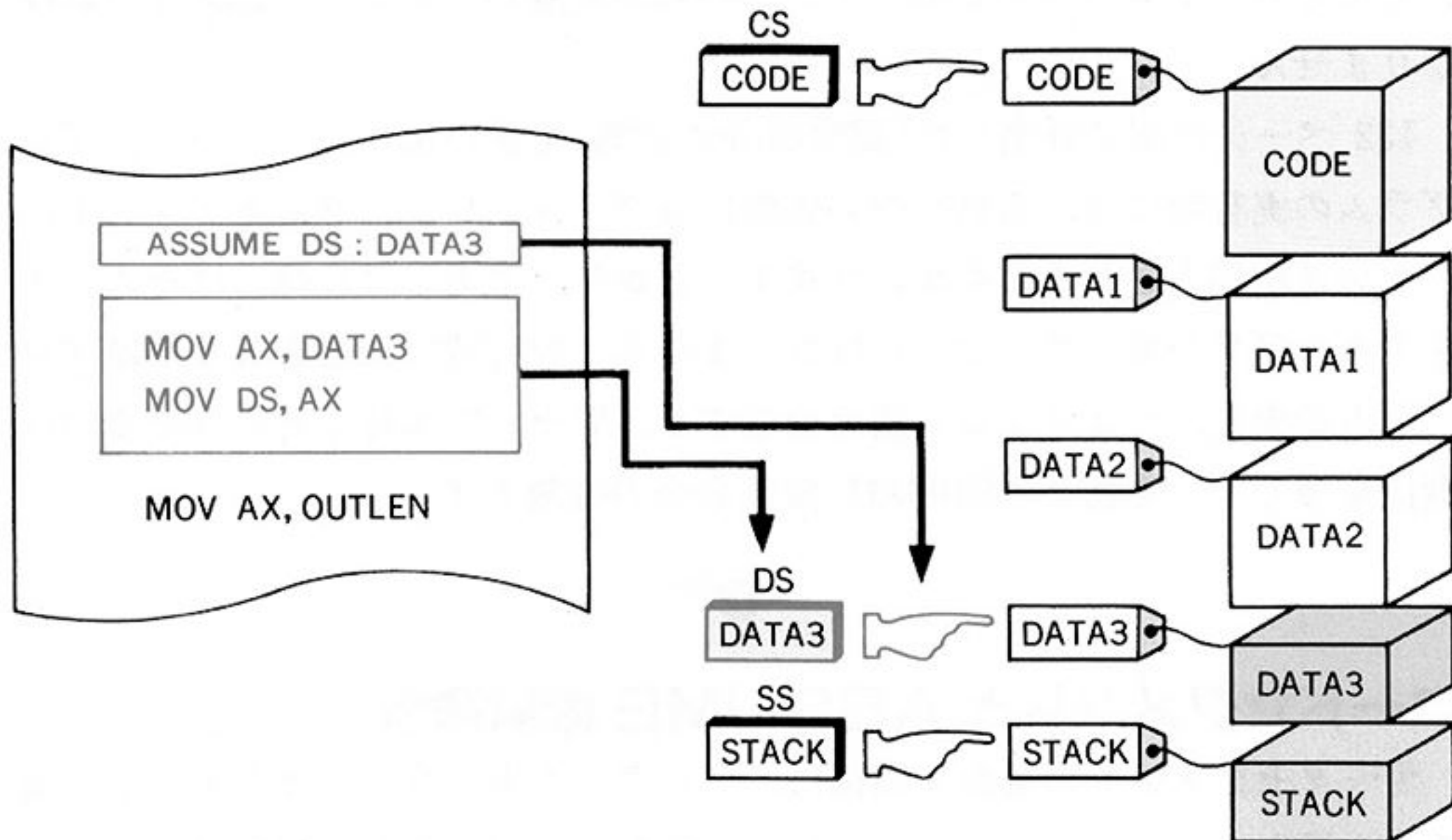
(a)

マシン語プログラムでセグメントDATA3のセグメントアドレスをDSレジスタにセットすることにより，セグメントDATA3をアクセスすることができるようになる．しかし，アセンブラはDSがセグメントDATA1を指していると認識しているので，正しいコードを出力することができない．

マシン語プログラムでDSレジスタにセグメントアドレスをセットするとともに，ASSUME擬似命令でアセンブラにもセグメントレジスタの内容に変更があることを知らせることによって，正しいコードが出力されるようになる．

図 4-21　セグメントの変更と ASSUME 擬似命令

データセグメントがいくつもある場合には，DS レジスタにセグメントアドレスをセットしなおすことで，どのセグメントにもアクセスすることができます．このとき注意しなければならないのは，セグメントレジスタに新しい値をセットするときには再び ASSUME 擬似命令が必要なことです．セグメントレジスタとセグメントの対応をあらためて MASM に教えなければならないのです．そうしないと，MASM は正しいコードを出力できません．

図 4-21-a のように DS レジスタにセグメント DATA1 のセグメントアドレスがセットされている状態から，DS レジスタの内容を変更して，セグメント DATA3 をアクセスする場合を考えます．このとき ASSUME 擬似命令を忘れていると，セグメント DATA3 にあるデータラベル OUT_LEN をアクセスしようとしても，MASM は DS レジスタがセグメント DATA1 を指していると仮定していますから，セグメント DATA1 に属さないデータラベル OUT_LEN のアドレスを求めることはできません．

このような場合には，図4-21-bのようにASSUME擬似命令でMASMにもセグメントレジスタとセグメントの対応が変わったことを伝えなければなりません．

132ページの図で解説した「擬似命令の役割」を思い出してください．プログラムの実行時には，当然ながら実際にセグメントレジスタにセグメントのアドレスが代入される必要があります．しかし，アセンブル時にはそのことをアセンブラが知っていなければなりません．実行時に対応させるのはプログラムの中身，つまりマシン語命令ですが，アセンブル時に対応させるのは擬似命令です．それがASSUME擬似命令の役割です．

コードセグメントとASSUME擬似命令

データセグメントの場合と同様に，プログラム中にコードセグメントが複数ある場合には，コードセグメントを定義するごとにASSUME擬似命令が必要です．

すべてのマシン語命令やコードラベルは，それが定義されたコードセグメントに属します．CSレジスタは1つ前のコードセグメントを指しているとMASMは仮定しているのに，新たなコードセグメントにマシン語命令を記述すると，MASMは正しいコードを出力できません．たとえば，ラベルを定義してそこへジャンプするという場合に，そのラベルは現在CSレジスタが指しているセグメントには属していないことになるので，アドレスを求めることができません(図4-22)．

```
A>TYPE UCASE.ASM
        ASSUME  CS:CODE1
CODE1   SEGMENT
START:
        MOV     AH,8
        INT     21H
        JMP     FAR PTR UCASE
OUTPUT:
        MOV     AH,2
        MOV     DL,AL
```

```
        INT     21H
        CMP     DL,'Z'-'A'+1
        JNE     START
        MOV     AH,4CH
        MOV     AL,Ø
        INT     21H
CODE1   ENDS

CODE2   SEGMENT

UCASE:
        CMP     AL,'a'
        JGE     CHK_Z
        JMP     FAR PTR OUTPUT
CHK_Z:
        CMP     AL,'z'
        JLE     TO_UCASE
        JMP     FAR PTR OUTPUT
TO_UCASE:
        AND     AL,5FH
        JMP     FAR PTR OUTPUT
CODE2   ENDS

STACK   SEGMENT STACK
        DB      1ØØH DUP (?)
STACK   ENDS
        END START

A>MASM UCASE;
Microsoft MACRO Assembler  Version 3.ØØ
(C)Copyright Microsoft Corp 1981, 1983, 1984

ØØØØ                          UCASE:
E r r o r  ---      62:No or unreachable CS
ØØØ9                          CHK_Z:
E r r o r  ---      62:No or unreachable CS
ØØ12                          TO_UCASE:
E r r o r  ---      62:No or unreachable CS

49698 Bytes free
Warning Severe
Errors  Errors
Ø       3

A>
```

CSをCODE2にASSUMEし直していないと……

……エラーが発生する

図4-22 コードセグメントの変更とASSUME擬似命令

4.5 GROUP 擬似命令

セグメントのグループ化

MASM では，いくつかのセグメントをまとめて1つのセグメントとして扱うことが可能です．便宜上いくつかのセグメントとして分けて定義しておいて，それらをアクセスする際には1つのセグメントとして扱うのです．

なぜこのようなことをするのかというと，次のような理由からです．プログラムで扱うデータは，種類によっていくつかの異なるセグメントに分割しておくと便利な場合がよくあります．たとえば，初期値を持つデータと持たないデータをそれぞれ別々のセグメントに定義しておくと，ソースプログラムのあちらこちらで定義してもオブジェクトプログラムではそれぞれのセグメントに集められます(129 ページ参照)．そうしておけばデバッグ時に初期値を探して変更することが容易に行えます．

しかし，1度にアクセスできるデータセグメントは DS レジスタによって指定されるセグメント1つだけです*．いくつものセグメントをアクセスしようとすると，異なるセグメントをアクセスするたびにセグメントレジスタにセグメントアドレスをセットしなおさなければなりません．これは非常に面倒である上に，余分な時間がかかるので実行速度も遅くなります．ですから，データセグメントは1つであることが望ましいのです．

GROUP 擬似命令を使ってセグメントをグループ化することで，以上のような相反する望みをともに実現することができます．セグメントのグループ化とは，複数のセグメントを連結して1つのセグメントにすることで，連結してできるセグメントを**セグメントグループ**と呼びます．

---

* 142 ページで解説するセグメントオーバーライドプリフィックスを使えば，CS，ES，SS レジスタの指すセグメントをデータセグメントとしてアクセスすることも可能．

GROUP擬似命令の書式

［書式］ セグメントグループ名 GROUP セグメント名，セグメント名…

セグメントDATA1とDATA2をグループ化したセグメントグループDGROUPを定義するには，

```
DGROUP     GROUP     DATA1, DATA2
```

と宣言します．「GROUP」の左側にセグメントグループの名前を，そして右側に連結するセグメント名を並べて書きます．

なお，いくつかのセグメントを連結してできるセグメントグループも，1つのセグメントであることには違いありませんから，その大きさは64Kバイト以内でなければなりません．

セグメントグループとASSUME擬似命令

セグメントグループは通常のセグメントとまったく同様に扱うことができます．すなわち，セグメントグループの名前をセグメントアドレスとして用いDSレジスタにセットすることで，セグメントグループをデータセグメントとしてアクセスすることができます．もちろん，ASSUME擬似命令でMASMにもセグメントレジスタとセグメントの対応を伝えておかなければなりません(次ページの図4-23)．

セグメントグループに連結されている各セグメント内で定義されているラベルは，セグメントグループにも属することになります．たとえば図4-23のような場合，セグメントDATA2に属するラベルはセグメントグループDGROUPにも属するとみなします．

すると，そのラベルは2つのオフセットアドレスを持つことになります．1つはセグメントDATA2を単独のセグメントとしたときのオフセットアドレスであり，もう1つはセグメントグループDGROUPにおけるオフセットアドレスです．この違いを図4-24に示しました．

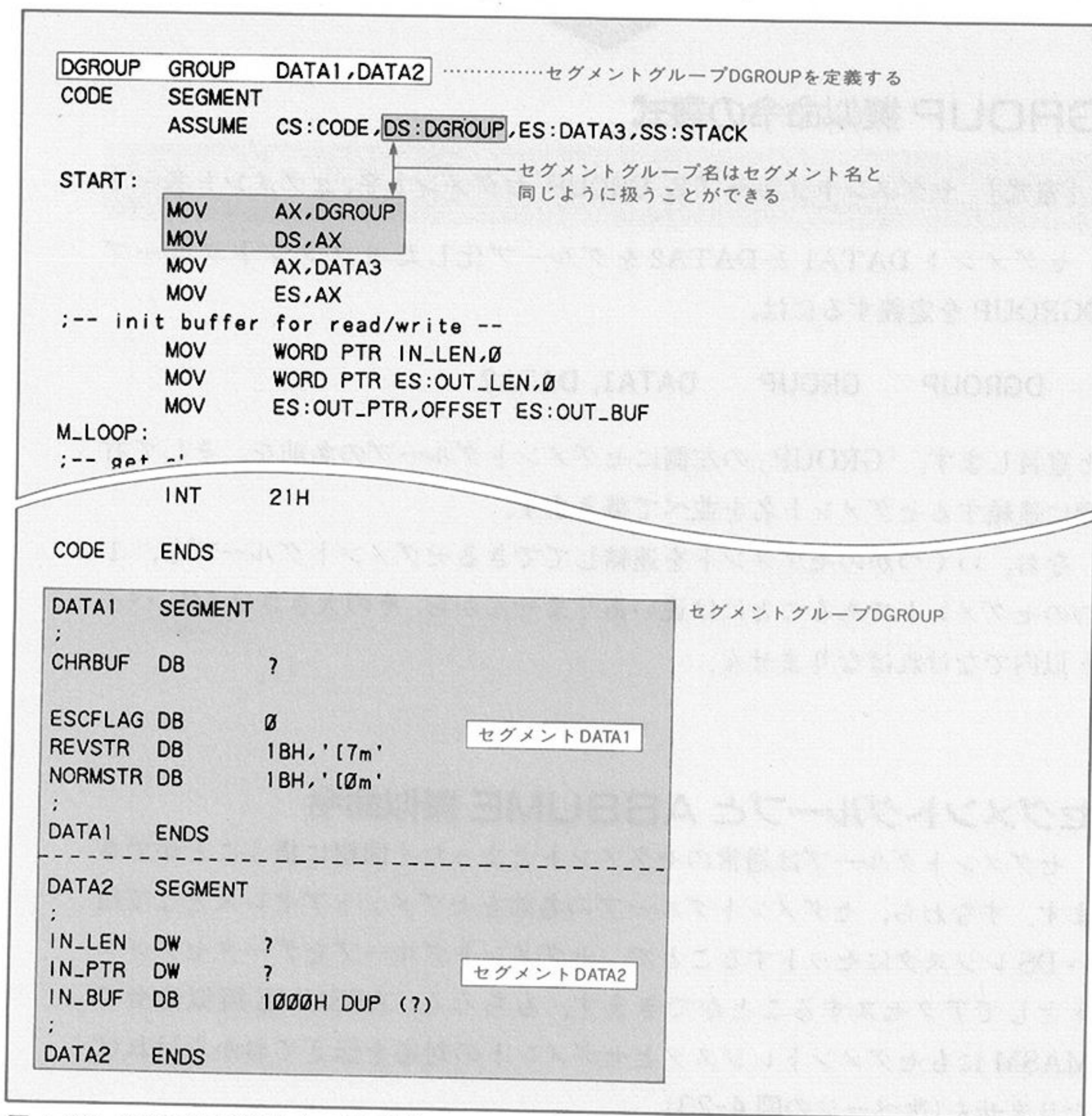



図 4-23　GROUP 擬似命令の使用例

DSレジスタにセグメントグループDGROUPのアドレスをロードし，ASSUME擬似命令でDSレジスタがDGROUPを指しているように指定します(図 4-24-a)．すると，データラベルIN_LENをアクセスするマシン語命令では，オフセットアドレスとしてDGROUP先頭からのアドレスが生成されます．

これに対し，セグメントDATA2を単独のセグメントとして扱い，DSレジスタにそのアドレスをロードし，ASSUME擬似命令でもDSレジスタが

DATA2を指しているように指定することもできます(図4-24-b). これは今まで解説したセグメントの扱い方そのままであり, データラベルIN_LENをアクセスするマシン語命令ではオフセットアドレスとしてDATA2先頭からのアドレスが生成されます.

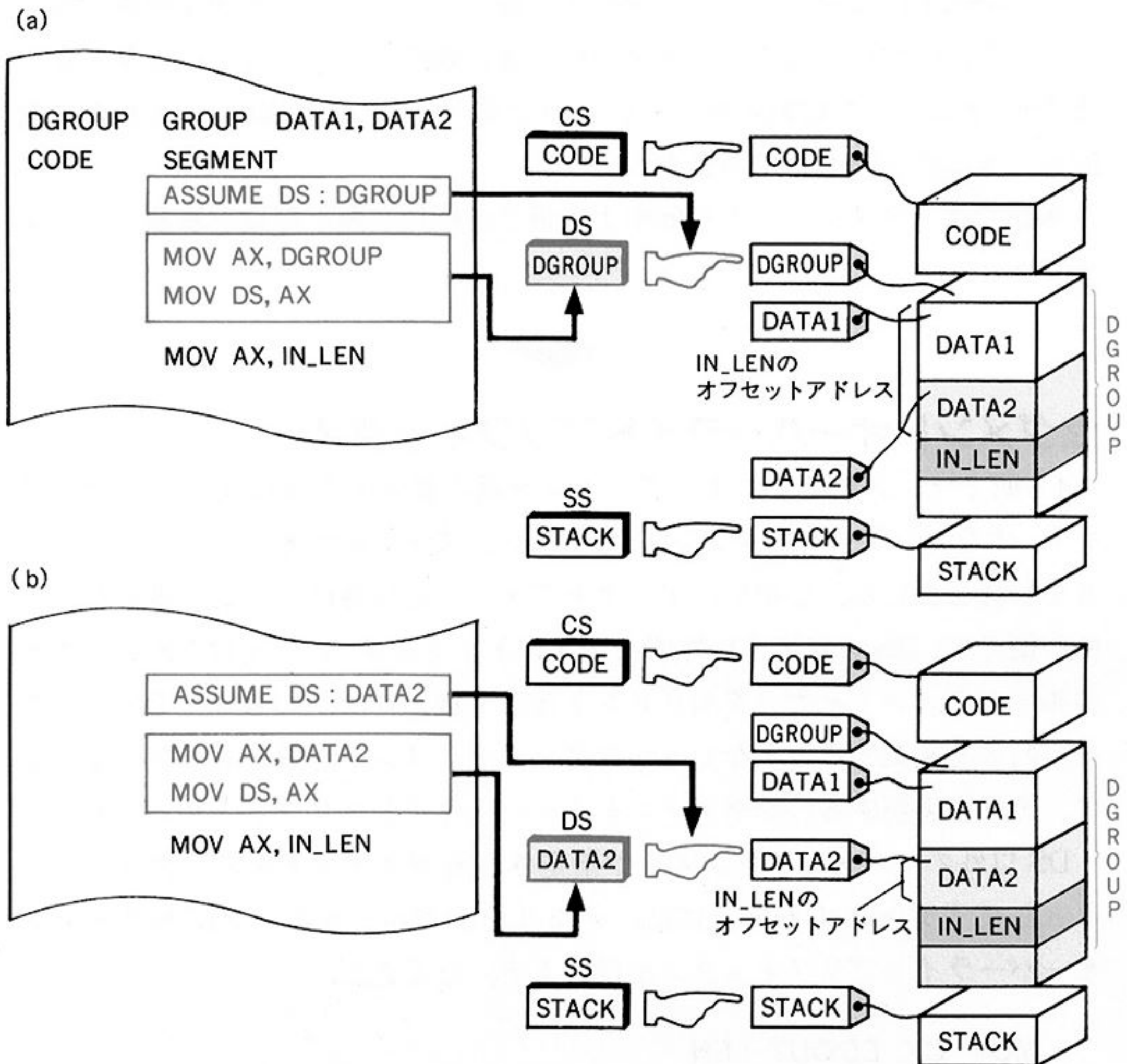
つまりASSUME擬似命令を使って, セグメントレジスタとセグメントの対応をMASMに教えることにより, このような区別が可能になっているのです.

GROUP擬似命令により, DATA1とDATA2という2つのセグメントをつなげて1つのセグメントとして扱うことができる. 上の図のようにそれぞれを1つの独立したセグメントとして扱うこともできる. 両者では同じラベルであってもオフセットアドレスが異なることに注意.

図4-24 セグメントグループとASSUME擬似命令

4.6 セグメントを使いこなす

セグメントの基本的な概念やその利用法について，これまで解説してきました．8086CPUはセグメントを効率よく扱うためのさらに便利な機能を持っています．たとえば，ESレジスタの使い道に疑問を感じた人もいるかと思いますが，実は大切な役割を持っているのです．そしてMASMでもそれらの機能をフルにサポートしています．

本節ではセグメントをより効率よく扱うために必要な知識を解説していきます．

セグメントオーバーライドプリフィックス

4.1節でマシン語命令によってメモリを読み書きする際に使われるセグメント，すなわちデータセグメントは，DSレジスタの指すセグメントであることを解説しました．しかし，データセグメントを同時に1つしか扱えないのは不便です．DSレジスタ以外のセグメントレジスタ，たとえばESレジスタの指すセグメントをデータセグメントとして扱えれば，いちいちDSレジスタの内容を変更することなく2つのデータセグメントを扱えることになります．それを実現するのがセグメントオーバーライドプリフィックスです．

DS以外のセグメントレジスタで指定されるセグメントをデータセグメントとしてアクセスする場合には，メモリを示すニーモニックにセグメントオーバーライドプリフィックスを付けます．たとえば，

```
MOV  CX, ES:OUT_LEN
```

という命令では，ESレジスタで指定されるセグメントのオフセットアドレスOUT_LENのメモリの内容がCXレジスタに転送されます．「ES:」の部分がセグメントオーバーライドプリフィックスです．このようにセグメントレ

ジスタの名前に「：」(コロン)を付けたものをセグメントオーバーライドプリフィックスと呼び，アドレス指定の前に付けておくことで，指定したセグメントレジスタの指すセグメントをデータセグメントとすることができます．このことを図で示したのが次の図 4-25 です．

ESレジスタはエキストラ(臨時の)データセグメントという名前が示すように，データのセグメントが2つ以上あってDSレジスタだけでは不便な場合に，セグメントオーバーライドプリフィックスを利用することによって用いられるレジスタです*.

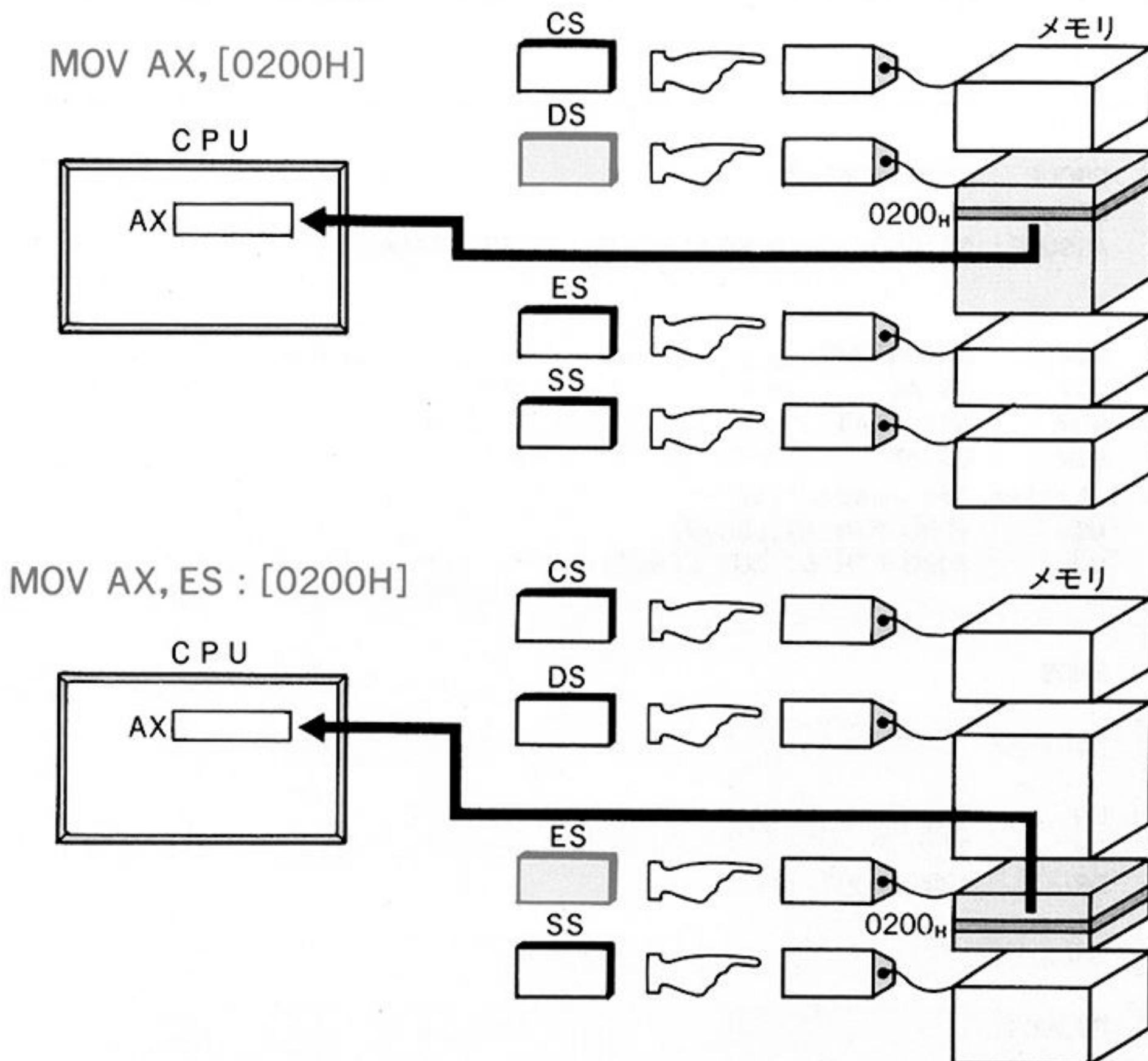



図 4-25　セグメントオーバーライドプリフィックスの役割

＊ES レジスタはストリング命令でも利用される．150ページのコラム参照．

同様にCS，SSレジスタで指定されるセグメントも，「CS:」，「SS:」とプリフィックスを付けることでデータセグメントとしてアクセスすることができます．ただし，マシン語コードやスタック領域とデータを混在させることになるので注意が必要です．

セグメントオーバーライドプリフィックスの自動挿入

セグメントオーバーライドプリフィックスを利用して，ESレジスタの指すセグメントをデータセグメントとして利用する場合にも，ASSUME擬似命令によってESレジスタとセグメントの対応をMASMに伝えておかなけ

図4-26 ESレジスタに対するASSUME擬似命令

ればなりません．その理由は4.2節で解説した通りです．

また，MASMにセグメントレジスタとセグメントの対応を伝えておくことにより，不注意によるプログラムミスを未然に防ぐことができます．

前ページの図4-26は本章の最初に紹介したESC.COMを改良したプログラムの一部です．DATA1，DATA2，DATA3という3つのデータセグメントがあり，DATA1とDATA2を連結してセグメントグループDGROUPを定義しています．DS，ESレジスタはそれぞれDGROUP，DATA3のセグメントを指すようにASSUME擬似命令で指示し，マシン語プログラムでもそのとおりにレジスタをセットしています．このとき，

```
MOV  WORD  PTR  ES:OUT_LEN, 0
```

という命令でセグメントオーバーライドプリフィックス「ES:」を付け忘れるとどうなるでしょうか．セグメントDATA3のオフセットアドレスOUT_LENをアクセスするはずが，セグメントグループDGROUPのオフセットアドレスOUT_LENをアクセスしてしまいます．このため次の図4-27に示すようなエラーが発生します．これは，MASMのセグメントオーバライドプリフィックスの自動挿入機能が働いた結果で，以下に示すようなアセンブル過程を知ることにより，その原因を理解することができます．

MASMのアセンブルは2パスで行われます．パス1，つまり1回目のアセンブルでは各命令やデータの必要とするバイト数からラベルに対応するアド

```
A>MASM ESC;
Microsoft MACRO Assembler  Version 3.00
(C)Copyright Microsoft Corp 1981, 1983, 1984

                               M_LOOP:
Error   ---          6:Phase error between passes

49212 Bytes free

Warning Severe
Errors  Errors
0       1

A>
```


図4-27 フェイズエラーの例

レスを決定します．そしてパス 2，つまり 2 回目のアセンブルであらためてそのアドレス値を使って各命令をアセンブルします．

パス 1 ではエラーの原因となったマシン語命令をアセンブルする際に OUT_LEN というデータラベルは定義されていないので，それがどのセグメントに属するのかわかっておらず，セグメントオーバーライドプリフィックスは必要ないものとしてアセンブルします．ところが，パス 2 ではデータラベル OUT_LEN が ES レジスタの指すセグメントに属することがわかっていますから，セグメントオーバーライドプリフィックスを自動的に挿入してしまいます．セグメントオーバーライドプリフィックスは 1 バイトのマシン語命令ですから，パス 1 のときよりも 1 バイトずつ以降のアドレスがずれてしまいます．すると，それ以降のラベルの定義があるとパス 1 のときと対応するアドレスが食い違ってしまいます．

MASM はパス 1 とパス 2 でラベルのアドレスが食い違ってしまうことを「PHASE ERROR」（フェイズエラー）と呼び，そこでアセンブルを中断してしまいます（前ページの図 4-27）．「ES:」の付け忘れはこのようなエラーの原因となってしまうので，不注意によるミスを発見しやすくなるのです．

MASM のセグメントオーバーライドプリフィックスの自動挿入機能は，OUT_LEN が ES レジスタの指すセグメントに属していることを MASM がすでに知っている場合に，「ES:」を自動的に挿入してくれるという機能です．OUT_LEN がセグメント DATA3 内で定義されていることがわかっており，ASSUME 擬似命令で ES レジスタとセグメント DATA3 の対応を指示してあれば，この機能が働きます．

つまり，データセグメントの定義をマシン語命令で参照されるよりも前に行えば，「ES:」の指定を省いてしまうことができます．MASM にセグメントに関する情報をすべて与えてからマシン語命令を記述することで，プリフィックスの自動挿入機能を有効に活用することが可能です．

前方参照をしないようにするか，セグメントオーバーライドプリフィックスをきちんと付けるかのいずれかの方法をとるかは，プログラマにまかされています．しかし，定義の順番を気にするよりも必要なセグメントオーバーライドプリフィックスを忘れずに自分で書いておくことをお勧めします．どの

セグメントにあるデータにアクセスするのかを常に自分で把握していなければ，思わぬバグになやまされることになりかねません．そのためにもセグメントレジスタがどのセグメントを指しているかをしっかり意識して，必要なセグメントオーバーライドプリフィックスをソースプログラムに記述した方が確実です．後でプログラムを見直すときにも，セグメントの構成がつかみやすくなります．

暗黙のセグメント指定　（ローカル変数）

セグメントオーバーライドプリフィックスを付けなければ必ずDSレジスタの指すセグメントがデータセグメントになるかのように説明してきましたが，実は必ずしもそうではなく若干の例外があります．

それは，BPレジスタをポインタとして使うアドレッシングモードでは，DSレジスタではなくSSレジスタで指定されるセグメントがデータセグメントとなることです．たとえば，

```
MOV  AX, [BP]
```

という命令は，SSレジスタで指定されるセグメントのBPレジスタで指定されるアドレスのメモリの内容をAXレジスタに転送するという命令です(図4-28-a)．

```
MOV  DX, [BP+4]
MOV  BX, [BP+SI+6]
```

などの命令も同様です．

DSレジスタで指定されるセグメントを対象としたい場合は，逆に「DS:」というセグメントオーバーライドプリフィックスが必要となります(図4-28-b)．

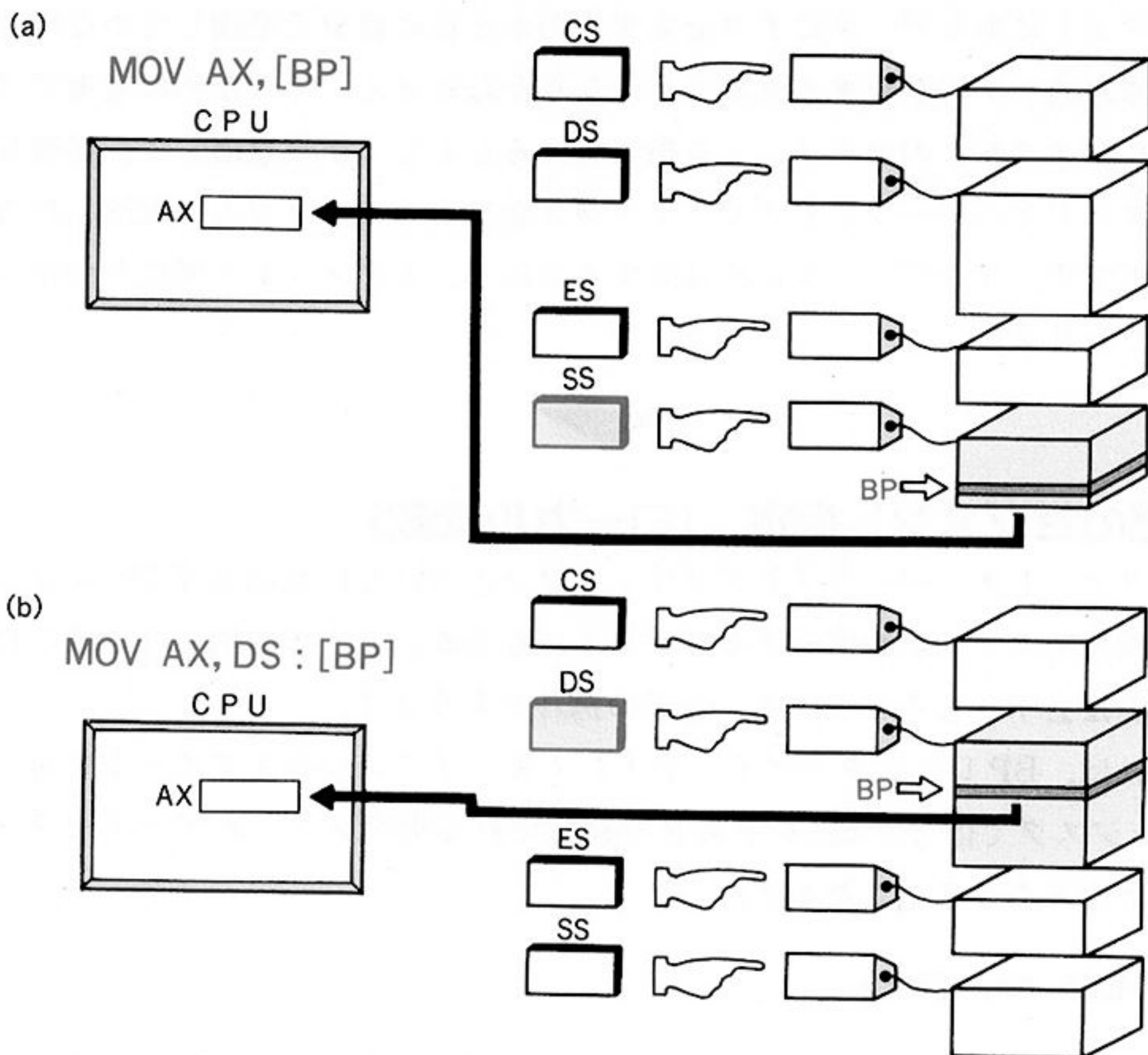



図 4-28　BP レジスタを使ったアドレッシングモードとスタックセグメント

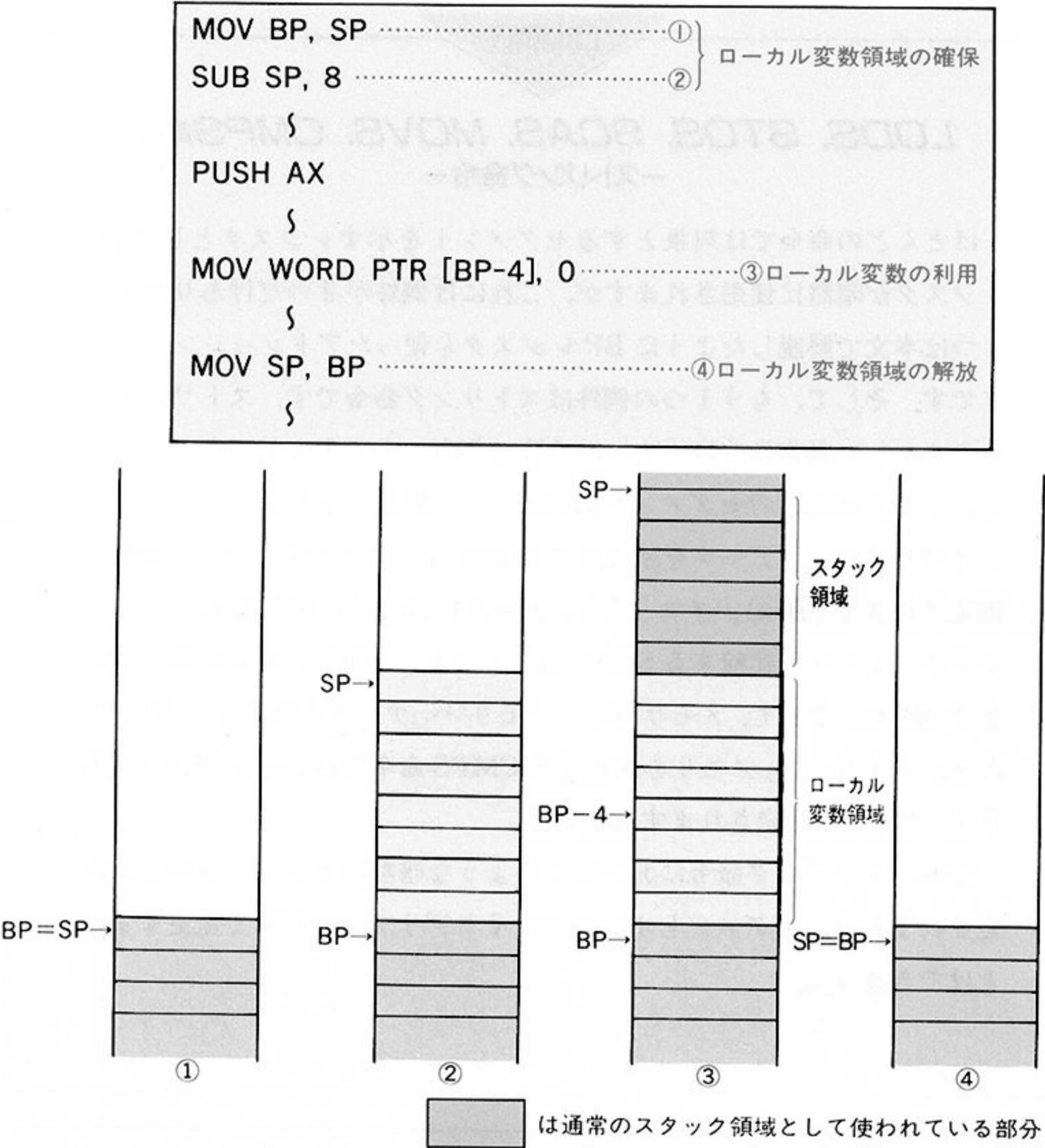
BP レジスタは，スタック領域の一部を一時的にワークエリアとして確保するために使われるレジスタです．具体的には次の図 4-29 に示す手順でスタック領域の一部をワークエリアとして確保することができます．

まず，SP レジスタ(スタックポインタ)の内容を BP レジスタに転送します(図 4-29 ①)．そして SP レジスタを適当な数だけ減らします(図 4-29 ②)．スタック領域は上位から下位へと使われていきますから，その数だけ空白地帯ができることになります．その領域をワークエリアとして利用するのです．こうして確保したワークエリアを「ローカル変数領域」と呼びます．図 4-29 では 8 バイト分の領域をスタック領域から確保したことになります．

ローカル変数領域は SS レジスタの指すセグメントにあるので，BP レジスタをポインタとするアドレッシングモードでアクセスします(図 4-29 ③)．

図 4-29 では 4 個のワード型データとして利用しています．もちろん，PUSH 命令などでさらにスタック領域を利用することもできます．この場合はローカル変数領域よりもさらに下位のメモリが継続してスタックとして使われることになります(図 4-29 ③)．

ローカル変数領域を確保するというテクニックは，サブルーチンなどで一時的にしか必要としないデータ領域を確保するために使われます．必要がなくなれば，その時点で解放することができるからです(図 4-29 ④)．通常の

図 4-29　ローカル変数領域の確保

データセグメントに確保してしまうと，特定のサブルーチンのための領域を常に占有してしまうことになりメモリの効率がよくありません．

以上の解説のようにBPレジスタはローカル変数領域へのポインタとして利用されるレジスタなので，「MOV DX, [BP−4]」のような命令では，SSレジスタの指すセグメントがデータセグメントとしてアクセスされることを覚えておいてください*．このことを「暗黙のセグメント指定」と呼んでいます．

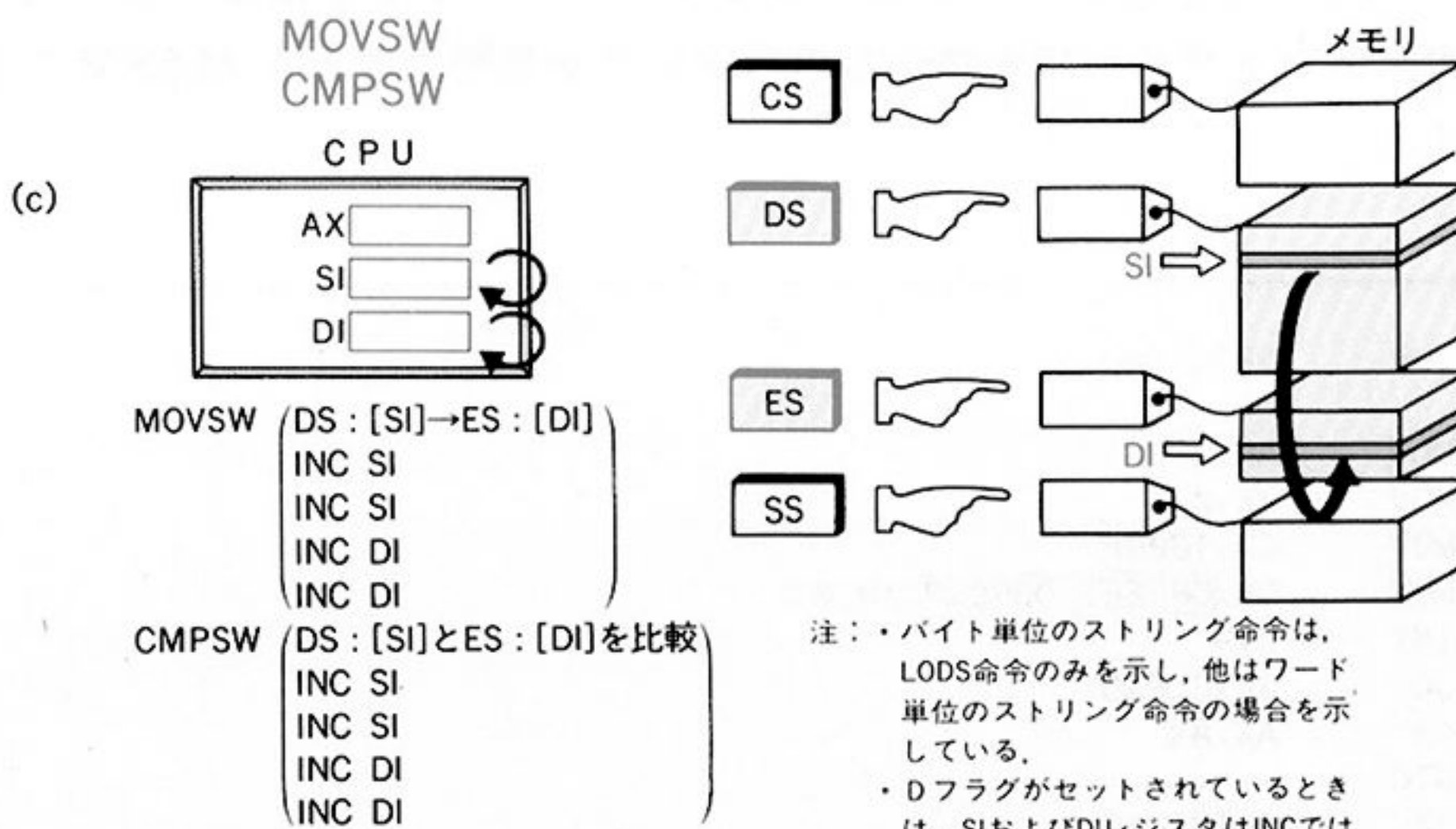
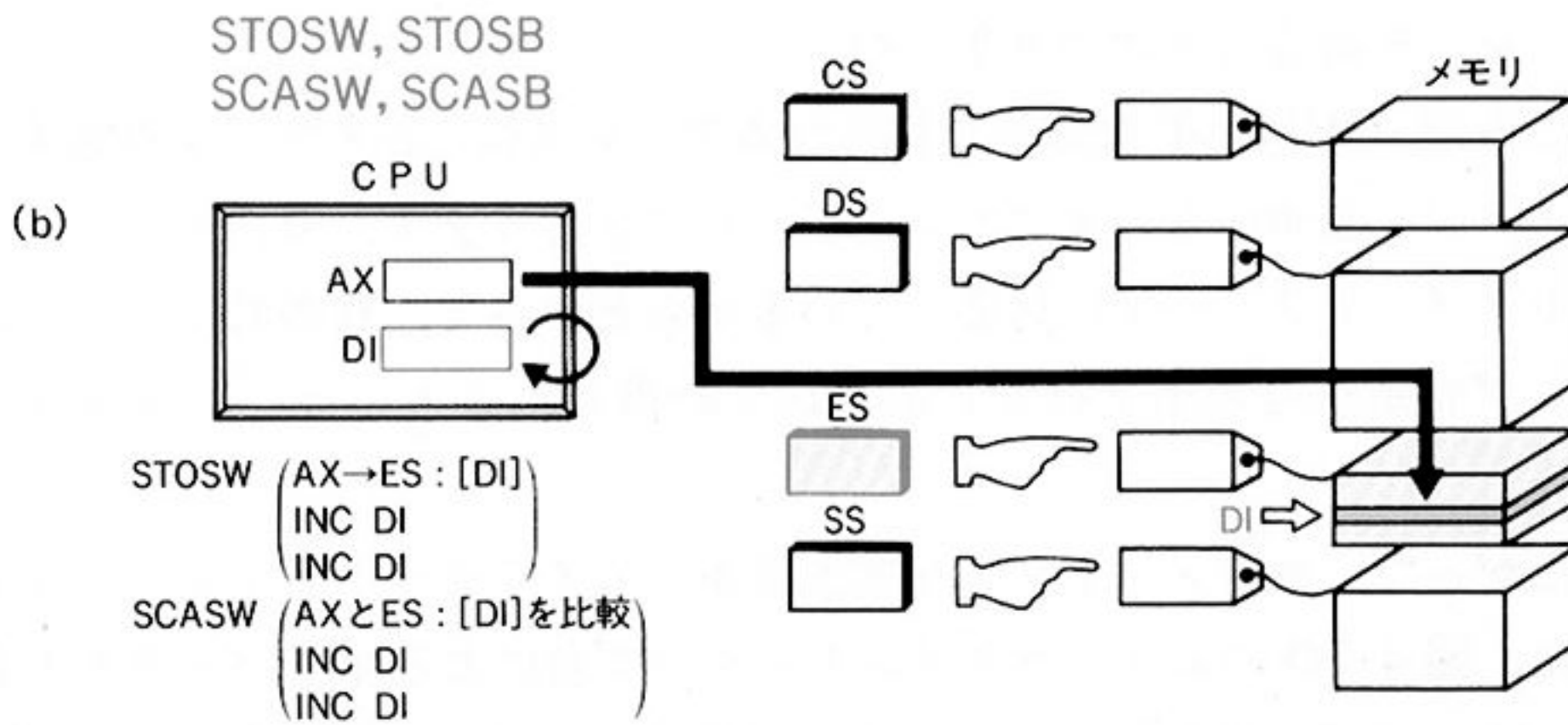
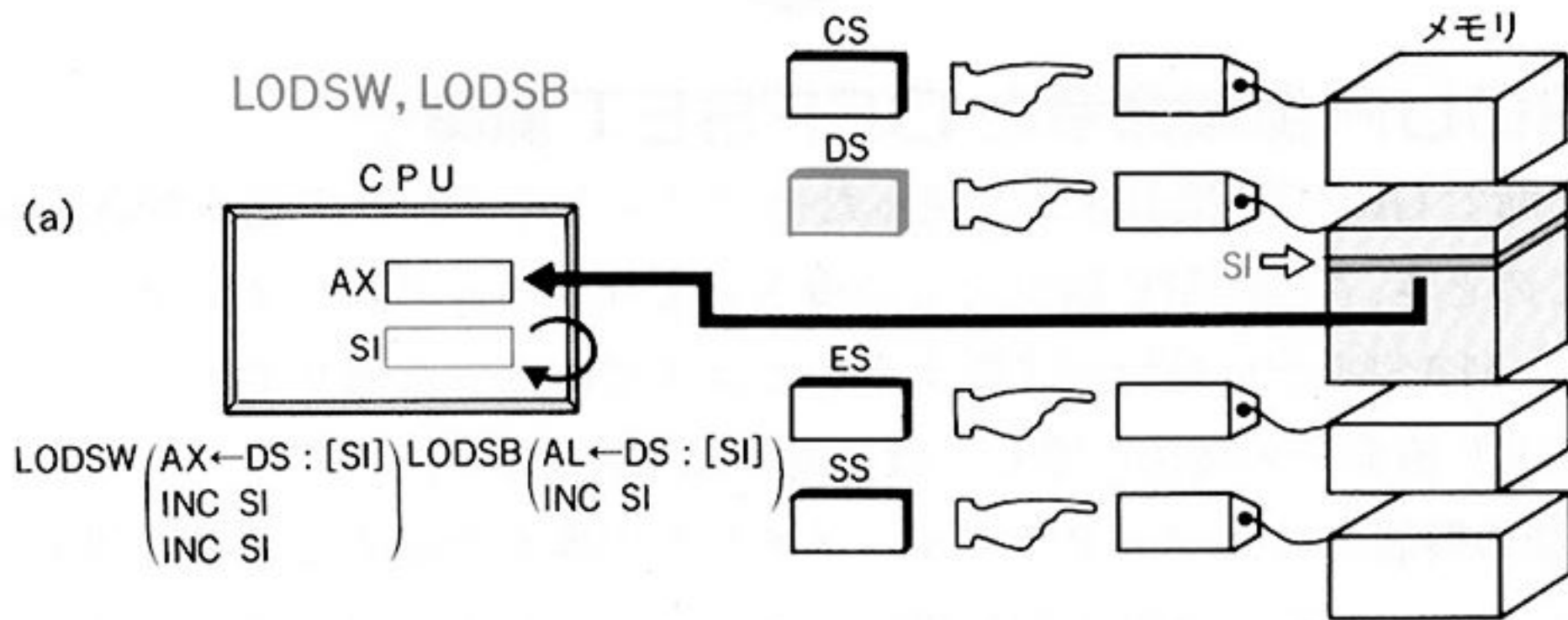
COLUMN

LODS, STOS, SCAS, MOVS, CMPS命令
ーストリング命令ー

ほとんどの命令では対象とするセグメントを示すレジスタとしてDSレジスタが暗黙に使用されますが，これには例外が2つだけあります．1つは本文で解説したようにBPレジスタを使ったアドレッシングモードです．そして，もう1つの例外はストリング命令です．ストリング命令のくわしい動作は前書「はじめて読む8086」を参考にしてもらうとして，ここでは暗黙のセグメント指定について解説しましょう．

メモリ「から」データを読み出すLODS命令ではDS: [SI] が暗黙に指定されます(図-a)．メモリ「へ」データを格納するSTOS命令，メモリ「と」データを比較するSCAS命令ではES: [DI] が暗黙に指定されます(図-b)．そして，メモリ「から」メモリ「へ」データを転送するMOVS命令，メモリ「と」メモリを比較するCMPS命令では，DS: [SI] → ES: [DI] が暗黙に指定されます(図-c)．

なお，ストリング命令におけるこのような暗黙のセグメント指定は固定されており，セグメントオーバーライドプリフィックスで変更することはできません．

*くわしくは6章で解説するが，ローカル変数領域はサブルーチンにパラメータを渡すためにも利用される．C言語とアセンブラをリンクさせる際には重要な概念となるのでよく理解しておくこと．

(a)
LODSW, LODSB
CPU
AX
SI
CS
DS
ES
SS
メモリ
SI
LODSW (AX←DS : [SI]
INC SI
INC SI)
LODSB (AL←DS : [SI]
INC SI)
(b)
STOSW, STOSB
SCASW, SCASB
CPU
AX
DI
CS
DS
ES
SS
メモリ
DI
STOSW (AX→ES : [DI]
INC DI
INC DI)
SCASW (AXとES : [DI]を比較
INC DI
INC DI)
(c)
MOVSW
CMPSW
CPU
AX
SI
DI
CS
DS
ES
SS
メモリ
SI
DI
MOVSW (DS : [SI]→ES : [DI]
INC SI
INC SI
INC DI
INC DI)
CMPSW (DS : [SI]とES : [DI]を比較
INC SI
INC SI
INC DI
INC DI)
注：・バイト単位のストリング命令は，
LODS命令のみを示し，他はワード
単位のストリング命令の場合を示
している．
・Dフラグがセットされているとき
は，SIおよびDIレジスタはINCでは
なくデクリメントされる．

GROUP 擬似命令と OFFSET 演算子

4.5 節で GROUP 擬似命令を使ったセグメントのグループ化を解説しました．このとき，ASSUME 擬似命令の働きによりラベルの持つオフセットアドレスを MASM が自動的に区別することはすでに述べた通りです．しかし，OFFSET 演算子の使用に際しては注意が必要です．

前節の解説では，データラベルをメモリの内容を参照するために使用していました．この場合，ASSUME 擬似命令でセグメントレジスタにグループ名を対応させることにより，自動的にセグメントグループ先頭からのオフセットアドレスを得ることができました．

ところが，OFFSET 演算子で得られるアドレスは，必ずラベルの定義されたセグメント先頭からのオフセットアドレスとなります．セグメントレジスタがセグメントグループに対応しているからといって，自動的にセグメントグループ先頭からのオフセットアドレスが得られるということはありません．

したがって，セグメントグループ先頭からのオフセットアドレスが必要な場合は，図 4-30 のようにセグメントグループ名によるオーバーライドを指定します．セグメントオーバーライドプリフィックスと同じように，データラベルにセグメントグループ名によるオーバーライドを指定することにより，セグメントグループ先頭からのアドレスが必要なことを MASM に指示します．

```
        JMP     GETC_END
GETC_1:
        MOV     AH,3FH
        MOV     BX,0
        MOV     CX,1000H
        MOV     DX,OFFSET DGROUP:IN_BUF
        INT     21H
        JC      GETC_END
        OR      AX,AX
        STC
        JZ      GETC_END
```

図 4-30　セグメントグループと OFFSET 演算子

EXEモデルのプログラム実習

これまで8086CPUに特有のセグメントの概念，そしてセグメントをMASMで扱う方法を解説してきました．セグメントを扱うにはたくさんの知識を必要とするように思えるかもしれませんが，すべてはセグメントレジスタの役割につながります．その点を理解していれば決して難しくはありません．

セグメントの概念を理解することにより，多くのメモリを自由に扱えるようになりました．いよいよ本章の冒頭で紹介したエスケープシーケンス生成プログラムESC.COMを，複数のセグメントを利用するESC.EXEに改良します．このプログラムを通して，EXEモデルのプログラムを作成するために必要な知識を確認するとともに，各擬似命令の効果を実際に調べてみましょう．

標準入出力をバッファリングするプログラム

改良後のプログラムESC.EXEのソースプログラムをリスト4-2に示します．

リスト4-2 エスケープシーケンスジェネレータ ESC.ASM（EXE版）

```
DGROUP  GROUP   DATA1,DATA2 ……………セグメントのグループ化を宣言
CODE    SEGMENT ……………セグメントの始まりを宣言
        ASSUME  CS:CODE,DS:DGROUP,ES:DATA3,SS:STACK ……………セグメントレジスタと
                                                              セグメントとの対応を
                                                              宣言
START:
        MOV     AX,DGROUP          ┐
        MOV     DS,AX              │ セグメントレジスタの
        MOV     AX,DATA3           │ 初期化
        MOV     ES,AX              ┘
;-- init buffer for read/write --   ┐
        MOV     WORD PTR IN_LEN,Ø  │ 入出力バッファの
        MOV     WORD PTR ES:OUT_LEN,Ø │ 初期化
        MOV     ES:OUT_PTR,OFFSET ES:OUT_BUF ┘
```

```
M_LOOP:                          このルーチンではファイルの終わりに
;-- get char -- ··············達したらキャリーフラグをセットする
        CMP     WORD PTR IN_LEN,Ø    } バッファに読み込まれて
        JE      GETC_1               } いなければGETC_1へ
        MOV     DI,IN_PTR            } バッファから1文字取り出す
        MOV     AL,[DI]              } バッファポインタを進め,
        INC     WORD PTR IN_PTR      } 残り文字数を減らす
        DEC     WORD PTR IN_LEN      }
        CLC ··············キャリーフラグをクリアする
        JMP     GETC_END
GETC_1:
        MOV     AH,3FH
        MOV     BX,Ø
        MOV     CX,1ØØØH                 } バッファにデータを
        MOV     DX,OFFSET DGROUP:IN_BUF  } 読み込む
        INT     21H
        JC      GETC_END ··············キャリーフラグがセットされて
                                       いれば読み込みエラー
        OR      AX,AX        } 読み込んだバイト数が0ならば,
        STC                  } ファイルの終わりに達した
        JZ      GETC_END     } キャリーフラグをセットする
        DEC     AX
        MOV     IN_LEN,AX                       } 読み込んだバイ
        MOV     AL,IN_BUF                       } ト数, バッファ
        MOV     IN_PTR,OFFSET DGROUP:IN_BUF+1   } ポインタをセッ
        CLC                                     } ト, バッファか
GETC_END:                                       } ら1文字取り出
;-- get char end --                               す
                                                  キャリーフラグ
                                                  をクリア
        JNC     ESC        ファイルの終わりに達したら, 出力バッファに
        JMP     FLUSH ······残されたデータを出力して終了へ

                                   (バッファリング機能を持つ1文字入力)

ESC: ··············このルーチンでは出力する文字列のアドレスをBXレジスタ,
;-- transfer routine --       文字数をCXレジスタにセットする
        CMP     ESCFLAG,BYTE PTR 1  } 反転中かどうかを調べる
        JE      NORMCHK             }
        CMP     AL,'[' ··············反転を開始するかどうか調べる
        JNE     THROUGH ··············'['でなければそのまま出力へ
;-- start reverse output --
        MOV     ESCFLAG,BYTE PTR 1  }
        MOV     BX,OFFSET REVSTR    } 反転を開始する
        MOV     CX,4                }
        JMP     PUTS                }
NORMCHK:
;-- check reverse end ? --
        CMP     AL,']'       } 反転中なら,
        JNE     THROUGH      } 反転を終了するかどうか調べる
;-- end reverse output --
        MOV     ESCFLAG,BYTE PTR Ø  }
        MOV     BX,OFFSET NORMSTR   } 通常の表示に戻す
        MOV     CX,4                }
        JMP     PUTS                }
THROUGH:
                                   (変換処理)
```

セグメントCODE

```
;-- output through --
        MOV     CHRBUF,AL           入力した文字を
        MOV     BX,OFFSET CHRBUF    そのまま出力する
        MOV     CX,1
;-- escape sequence insertion end --

;-- put char -- ……………このルーチンでは，出力に失敗すると，
PUTS:                       キャリーフラグをセットする
        CMP     CX,0                文字数が0になったら終了
        JE      PUTS_END
        MOV     AL,[BX]             表示する文字を取り出す
        INC     BX
        PUSH    BX
        PUSH    CX
;-- put char --
        CMP     WORD PTR ES:OUT_LEN,1000H   出力バッファに
        JE      PUTC_1                      文字が一杯かど
                                            うかを調べる
        INC     WORD PTR ES:OUT_LEN         出力バッファに文字を
        MOV     BX,ES:OUT_PTR               追加する
        MOV     ES:[BX],AL                  文字数を増やしバッファ
        INC     WORD PTR ES:OUT_PTR         ポインタを進める
        CLC ……………キャリーフラグをクリアする
        JMP     PUTC_END
PUTC_1:
        PUSH    AX
;-- buffer flush --
        PUSH    DS
        MOV     BX,ES
        MOV     DS,BX                       出力バッファが一杯で
        MOV     AH,40H                      あれば，バッファの
        MOV     BX,1                        内容をまとめて出力
        MOV     CX,ES:OUT_LEN               する
        MOV     DX,OFFSET ES:OUT_BUF
        INT     21H
        POP     DS
;-- buffer flush end --
        POP     BX
        JC      PUTC_END ……………キャリーフラグがセット
                                   されていれば出力エラー
        CMP     AX,ES:OUT_LEN       出力バッファの文字数と出力できた
        STC                         データ数が異なれば，出力エラー
        JNE     PUTC_END            キャリーフラグをセットする
        MOV     ES:OUT_BUF,BL                       出力バッフ
        MOV     WORD PTR ES:OUT_LEN,1               ァに文字を
        MOV     ES:OUT_PTR,OFFSET ES:OUT_BUF+1      格納する
        CLC ……………キャリーフラグをクリア
PUTC_END:
;-- put char end --
        POP     CX
        POP     BX                  変換ルーチンから渡された
        JC      QUIT                文字数分だけ繰り返す
        LOOP    PUTS
PUTS_END:
        JMP     M_LOOP ……………メインループ先頭へジャンプ
```

バッファリング機能を持つ1文字出力

```
;
FLUSH:
;-- remain output buffer ? --
        CMP     WORD PTR ES:OUT_LEN,Ø  } 出力バッファにデータが
        JE      QUIT                   } 残っているか？
;-- buffer flush --
        PUSH    DS
        MOV     BX,ES
        MOV     DS,BX
        MOV     AH,4ØH
        MOV     BX,1                     出力バッファに残っている
        MOV     CX,WORD PTR ES:OUT_LEN   データをまとめて出力する        終了処理
        MOV     DX,OFFSET ES:OUT_BUF
        INT     21H
        POP     DS
;-- buffer flush end --
QUIT:
        MOV     AH,4CH  } プログラムを終了する
        INT     21H     }

CODE    ENDS ……………セグメントの終わりを宣言
```

```
DATA1   SEGMENT
;
CHRBUF  DB      ? ……………通常出力用の仮バッファ

ESCFLAG DB      Ø ……………反転中かどうかを示すフラグ
REVSTR  DB      1BH,'[7m' ………表示を反転させるエスケープシーケンス
NORMSTR DB      1BH,'[Øm' ………表示を通常に戻すエスケープシーケンス
;
DATA1   ENDS
                                                  セグメントDATA1

DATA2   SEGMENT
;
IN_LEN  DW      ? ……………入力バッファ中の文字の数
IN_PTR  DW      ? ……………入力バッファから次に読み込むアドレス
IN_BUF  DB      1ØØØH DUP (?) ……………入力バッファ
;
DATA2   ENDS
                                                  DATA2セグメント
                                                  DGROUPセグメントグループ
```

```
DATA3   SEGMENT
;
OUT_LEN DW      ? ……………出力バッファ中の文字の数
OUT_PTR DW      ? ……………出力バッファへ次に書き込むアドレス
OUT_BUF DB      1ØØØH DUP (?) ……………出力バッファ
;
DATA3   ENDS
                                                  DATA3セグメント
```

```
STACK   SEGMENT STACK ……………スタックセグメントの定義
        DB      1ØØH DUP (?) スタック領域を十分な余裕をもって確保.
STACK   ENDS                 スタック領域はこのプログラムだけではなく,
                             各種割り込み処理でも勝手に使用されてしまうため.
                                                  STACKセグメント
        END START ……………ソースプログラムの終わりと実行開始
                             アドレスを宣言
```

ESC.COMを改良するにあたって入出力のバッファリングのテクニックを利用しました．データを1文字ずつ読み込んで処理するのではなく，一度に多くのデータを読み込んでまとめて処理したあと，やはり一度に出力するのです．

データの入力および出力には，MS-DOSの提供する便利な機能を利用しています．プログラムが起動された時点では**表4-1**に示すように，ファイルハンドル0番から4番に対応する5つのファイルが自動的にオープンされています．これらのファイルはオープン，クローズといった処理を省略していきなりアクセスすることができるのです*．

ファイルハンドル番号	名称	ファイル(デバイス)名	機能
0	標準入力	CON	通常コンソール画面(キーボード)が割りあてられているが，リダイレクトによりファイルに変更することもできる．
1	標準出力	CON	通常コンソール画面に割りあてられているが，リダイレクトによりファイルに変更することもできる．
2	標準エラー出力	CON	常にコンソール画面に割りあてられている．出力がリダイレクトされていても画面に出力したい場合に用いる．エラーメッセージなど注意を促したい出力に向いている．
3	標準補助装置	AUX	RS-232C等のデバイスに割り当てられている．入出力ともに可能．
4	標準プリンタ出力	PRN	プリンタに割り当てられている．

表4-1 MS-DOSであらかじめオープンされているファイル

これらのファイルのうち，ファイルハンドル0番と1番に割り当てられている「標準入力」と「標準出力」は，MS-DOSの重要な機能の1つです．これらは通常CONファイル，すなわちコンソール画面に割り当てられています．標準入力を読み出すことは，キーボードから入力を行うことになります．標準出力に出力することは，コンソール画面に表示することになります．MS

*ファイルハンドル，標準入出力等のMS-DOSに関する詳細は，「応用MS-DOS」(アスキー出版局)などを参考にするとよい．

-DOSのリダイレクト機能を使うと，入力と出力をファイルに切り替えることができますが，これは実は標準入力，標準出力を切り替える操作なのです．

このように標準入出力をリダイレクトすることにより，ファイル名を入力するなどのファイルオープンにともなう面倒な操作を省略することができます．また，ファイルをクローズする必要もありません．プログラムの終了と同時に自動的にクローズされます．

アセンブルの手順

アセンブル&リンクの手順は次の図4-31の通りです．COMモデルの場合と違ってEXE2BINコマンドの実行は必要ありません．アセンブルとリンクの作業だけでできるEXEファイルがそのまま実行可能なファイルとなります．

```
A>MASM ESC;
Microsoft MACRO Assembler  Version 3.00
(C)Copyright Microsoft Corp 1981, 1983, 1984


49212 Bytes free

Warning Severe
Errors  Errors
0       0

A>LINK ESC;

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

          No Stack Segmentの警告は出力されない

A>
```

図4-31　EXEモデルのアセンブル&リンク

なお，改良前のESC.COMを作成した人は，そのまま残しておくと改良後のプログラムであるESC.EXEを実行できません．COM型実行ファイルの方がEXE型実行ファイルよりも，実行の優先順位が高いからです．ESC.COMは削除しておいてください．

また，せっかく複数のセグメントを扱えるようにしたプログラムですが，どの環境でも動作するように入出力用のバッファとして 1000_H(4K) バイトしか確保していません．各自どれだけの大きさまで広げられるかを考えて，大きなバッファをとってみてください．それから，このプログラムは〝[]〟という記号自身や一部の漢字が使えないという欠点があります．ぜひ改良に挑戦してください．

図 4-31 では，リンクの時に「No Stack Segment」という警告が出ないことに注目しましょう．これは，EXE モデルでは自分でスタックセグメントを用意しているからです．COM モデルのプログラムではこの警告が出ても無視しましたが, EXE モデルでは警告が出るようだとプログラミングにミスがあることになります．

EXEモデルのプログラム実行開始時のセグメントレジスタ

プログラムがMS-DOSによって，ディスクからロードされ実行が開始された時点では，セグメントレジスタの内容はある約束に従って設定されています．EXEモデルのプログラムを作成するためには，この約束を知っている必要があります．

例題のプログラムESC.EXEのようなEXEモデルのプログラムの実行開始時には，図4-32に示すように各セグメントレジスタに各セグメントのアドレスが代入されています．

CSレジスタにはEND擬似命令で指定したプログラムの実行開始アドレスを含むセグメントCODEのセグメントアドレスがセットされています．もちろんIPレジスタにはそのオフセットアドレスがセットされています．COMモデルのプログラムではオフセットアドレス0100_H(に定義したラベ

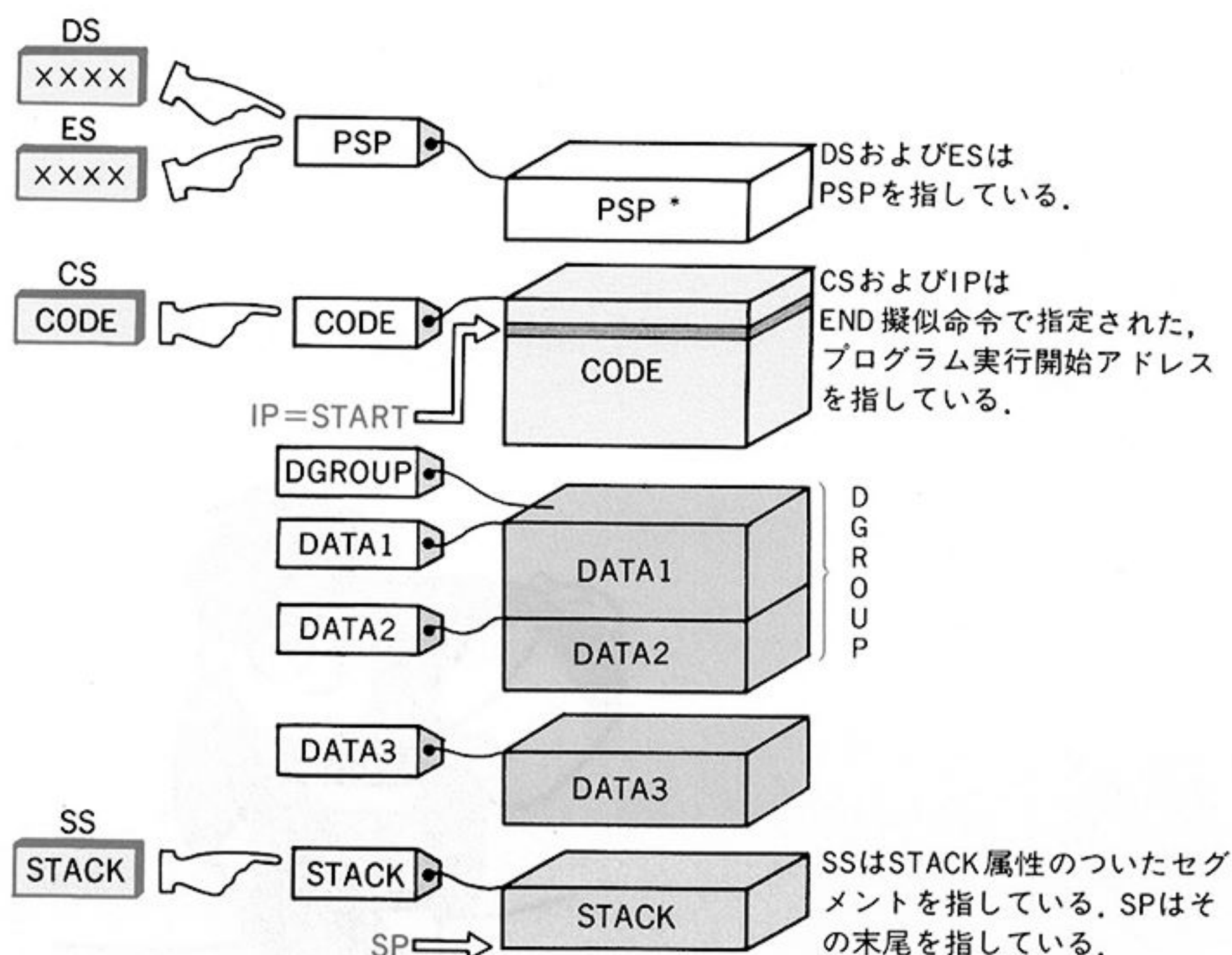



図4-32　EXEモデルのプログラム実行開始時のセグメントレジスタの内容

ル)に固定されていましたが，EXEモデルでは任意のアドレス(ラベル)をEND擬似命令で指定することができます(72ページの図3-7参照).

SSレジスタにはSTACK属性を付けたSTACKセグメントのセグメントアドレスがセットされています．そしてSPレジスタにはそのセグメントに属する最後のメモリのアドレス＋1，すなわち確保した領域の大きさがセットされています．スタックは下から上へと消費されていくことに注意してください．

DSおよびESレジスタにはデータを定義したセグメントのアドレスがセットされているわけではなく，PSPと呼ばれるセグメントのアドレスがMS-DOSによってセットされています．このためにデータセグメントをアクセスするためには，4.4節で解説したようにマシン語プログラムでセグメントレジスタにセグメントアドレスを代入しなければならなかったわけです(MS-DOSによって自動的に設定されない).

なお，PSPはCOMモデルのプログラムが実行される際にアドレス0100_Hまでの領域に用意されているものとまったく同一のもので，MS-DOSからプログラムに渡される各種の情報がセットされています(4.8節参照).

SEGMENT擬似命令，END擬似命令の効果

では，SEGMENT擬似命令やASSUME擬似命令など本章で解説した擬似命令の効果を確かめてみましょう．アセンブル&リンク後できあがったプログラムをSYMDEB(DEBUG)上で実行して，動作を確認します．

まず，プログラムがロードされた時点での各レジスタの内容を確認します．図4-33に示すように，CSおよびIPレジスタには，プログラムの先頭アドレスがセットされています．逆アセンブルしてみると，確かにラベルSTARTから始まるプログラムが現れます．END擬似命令で指定したSTARTというラベルの属するCODEセグメントのアドレスがCSレジスタに，そのオフセットアドレスがIPにセットされます．

SSレジスタにはスタックセグメントとして定義したSTACKセグメントのアドレスがセットされています．そして，SPレジスタにはスタック領域として確保したメモリの大きさがセットされています．

```
A>SYMDEB ESC.EXE
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-R
AX=0000  BX=0000  CX=2144  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=421F  ES=421F  SS=4444  CS=422F  IP=0000   NV UP EI PL NZ NA PO NC
422F:0000 B84142          MOV     AX,4241
-U 422F:0000
422F:0000 B84142          MOV     AX,4241
422F:0003 8ED8            MOV     DS,AX
422F:0005 B84343          MOV     AX,4343
422F:0008 8EC0            MOV     ES,AX
422F:000A C70610000000    MOV     Word Ptr [0010],0000
422F:0010 26C70600000000  MOV     Word Ptr ES:[0000],0000
422F:0017 26C70602000400  MOV     Word Ptr ES:[0002],0004
422F:001E 833E100000      CMP     Word Ptr [0010],+00
```


図 4-33　EXE モデルのプログラム起動時のレジスタの内容

次にプログラムをSYMDEBのトレース機能を使って少しずつ実行し，データセグメントがうまく割り当てられているかどうかを確かめてみます(図 4-34)．

```
-G 000A
AX=0000  BX=4343  CX=2144  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=4241  ES=4343  SS=4444  CS=422F  IP=000A   NV UP EI PL NZ NA PO NC
422F:000A C70610000000    MOV     Word Ptr [0010],0000         ;BR0 DS:0010=0000
-D 4241:0000
4241:0000  00 00 1B 5B 37 6D 1B 5B-30 6D 00 00 00 00 00 00   ...[7m.[0m......
4241:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-D 4343:0000
4343:0000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4343:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4343:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4343:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4343:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

```
4343:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4343:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4343:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-U 0 44↵
422F:0000 B84142          MOV     AX,4241
422F:0003 8ED8            MOV     DS,AX
422F:0005 B84343          MOV     AX,4343
422F:0008 8EC0            MOV     ES,AX
422F:000A C70610000000    MOV     Word Ptr [0010],0000
422F:0010 26C70600000000  MOV     Word Ptr ES:[0000],0000
422F:                             Word Ptr ES:[0002],0004
                                           [0010],+00
422F:0037 B43F            MOV     AH,3F
422F:0039 BB0000          MOV     BX,0000
422F:003C B90010          MOV     CX,1000
422F:003F BA1400          MOV     DX,0014
422F:0042 CD21            INT     21
422F:0044 7213            JB      0059
-G 0044↵ ……………………入力バッファに読み込むところまで実行する
[FUG] THE [BUG]↵ ……………プログラムが実行され入力待ちになるので，文字列を入力する
AX=0011  BX=0000  CX=1000  DX=0014  SP=0100  BP=0000  SI=0000  DI=0000
DS=4241  ES=4343  SS=4444  CS=422F  IP=0044   NV UP EI PL ZR NA PE NC
422F:0044 7213            JB      0059                         ;BR0
-D 4241:0000↵                   ←IN_BUFに格納されている
4241:0000  00 00 1B 5B 37 6D 1B 5B-30 6D 00 00 00 00 00 00   ...[7m.[0m......
4241:0010  00 00 00 00 5B 46 55 47-5D 20 54 48 45 20 5B 42   ....[FUG] THE [B
4241:0020  55 47 5D 0D 0A 00 00 00-00 00 00 00 00 00 00 00   UG].............
4241:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4241:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-U↵
422F:0046 0BC0            OR      AX,AX
422F:0048 F9              STC
422F:0049 740E            JZ      0059
422F:004B 48              DEC     AX
422F:004C A31000          MOV     [0010],AX
422F:004F A01400          MOV     AL,[0014]
422F:0052 C70612001500    MOV     Word Ptr [0012],0015
422F:0058 F8              CLC
-G 004F↵ ……………入力バッファから先頭の内容を取り出すところまで実行する
AX=0010  BX=0000  CX=1000  DX=0014  SP=0100  BP=0000  SI=0000  DI=0000
DS=4241  ES=4343  SS=4444  CS=422F  IP=004F   NV UP EI PL NZ NA PO CY
422F:004F A01400          MOV     AL,[0014]               ;BR1 DS:0014=5B
-T↵ …… 命令トレース                  IN_BUF
AX=005B  BX=0000  CX=1000  DX=0014  SP=0100  BP=0000  SI=0000  DI=0000
DS=4241  ES=4343  SS=4444  CS=422F  IP=0052   NV UP EI PL NZ NA PO CY
422F:0052 C70612001500    MOV     Word Ptr [0012],0015          DS:0012=0000
```

確かにDS＝DGROUP，ES＝DATA3となっていることがわかる．初期値を？で指定したものについては，ここでは0になっている

入力バッファ(IN_BUF)の先頭の内容をALレジスタに転送する．設定したとおりにセグメントが割り振られている

図 4-34　データセグメントの確認

以上の実験から，SEGMENT 擬似命令により定義したセグメントが指示通りに確保されていることがわかります．なお，セグメントアドレスの値はMS-DOS システムのバージョンやCONFIG.SYS ファイルによる設定によって変化しますから，みなさんが試す場合と異なるかもしれません．

ASSUME 擬似命令，GROUP 擬似命令の効果

ASSUME 擬似命令の効果を確かめるために，別の実験をやってみましょう．図 4-35 のようなプログラムを用意します．これをアセンブル&リンクして SYMDEB 上で実行してみます．

```
A>TYPE GTEST.ASM
DGROUP  GROUP   DATA1,DATA2
        ASSUME  CS:CODE
CODE    SEGMENT
START:
        MOV     AX,0

        ASSUME  DS:DGROUP
        MOV     AX,DGROUP
        MOV     DS,AX

        MOV     IN_LEN,AX

        ASSUME  DS:DATA2
        MOV     AX,DATA2
        MOV     DS,AX

        MOV     IN_LEN,AX

CODE    ENDS

DATA1   SEGMENT
ESCFLAG DB      0
REVSTR  DB      1BH,'[7m'
NORMSTR DB      1BH,'[0m'
DATA1   ENDS

DATA2   SEGMENT
IN_LEN  DW      ?
IN_PTR  DW      ?
IN_BUF  DB      1000H DUP (?)
DATA2   ENDS
```

```
STACK    SEGMENT STACK
         DB      100H DUP (?)
STACK    ENDS

         END START

A>SYMDEB GTEST.EXE
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-U 0000 0012
422F:0000 B80000         MOV    AX,0000
422F:0003 B83142         MOV    AX,4231
422F:0006 8ED8           MOV    DS,AX
422F:0008 A31000         MOV    [0010],AX
422F:000B B83242         MOV    AX,4232
422F:000E 8ED8           MOV    DS,AX
422F:0010 A30000         MOV    [0000],AX
```


図 4-35　ASSUME 擬似命令と GROUP 擬似命令の効果

ASSUME 擬似命令により DS レジスタにセグメント DATA2 を割り当てた場合には，ラベル IN_LEN は 0000_H というアドレスに置き換えられています．これに対しセグメントグループ DGROUP を割り当てた場合には，0010_H というアドレスに置き換えられています．同じラベルに対しても，セグメントレジスタの設定によって異なるアドレスが出力されていることがわかります．

前者はセグメント DATA2 の先頭をアドレス 0 としたときの IN_LEN のアドレスであり，後者はセグメント DGROUP の先頭をアドレス 0 としたときの IN_LEN のアドレスです．両者はセグメントアドレスおよびオフセットアドレスがそれぞれ異なりますが，結局は同じ物理アドレスを指しているのです(141 ページの図 4-24 参照)．

セグメントオーバーライドプリフィックスの確認

セグメントオーバーライドプリフィックスを付けた命令と付けていない命令が，それぞれどのようなマシン語コードと対応しているかを比べてみましょう．図4-36にSYMDEB(DEBUG)でディスアセンブルしてみた例を示します．

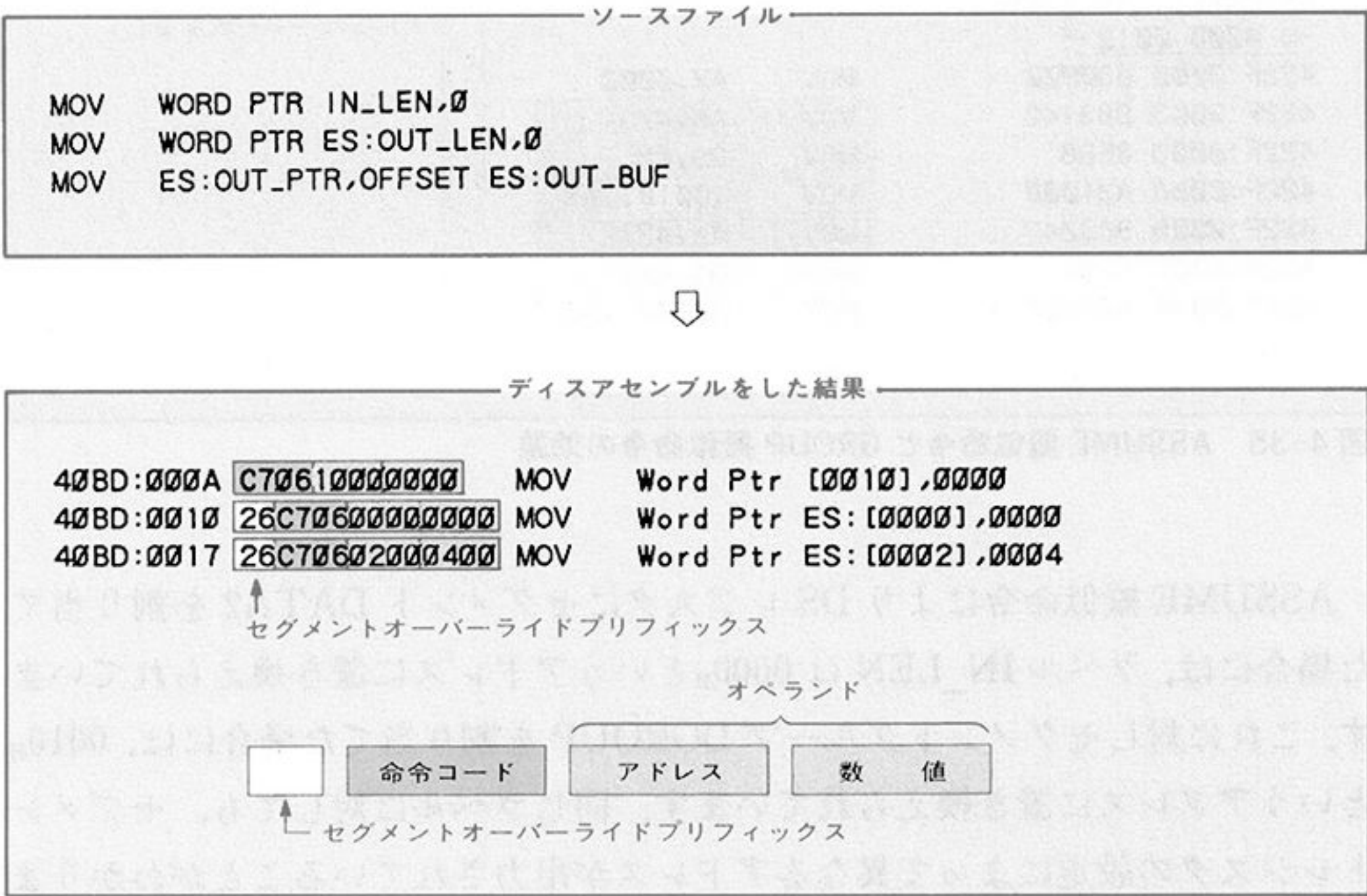



図4-36 セグメントオーバーライドプリフィックスのマシン語コード

セグメントオーバーライドプリフィックスの付いた命令は，1バイト余分なコードが前に付いていることがわかるでしょう．このコードは次の命令はメモリのアクセスにESレジスタを使う，ということをCPUに指示する1種の命令なのです．正確にはこの1バイトのコードのことをセグメントオーバーライドプリフィックスと呼びます．

セグメントのリロケート

SEGMENT擬似命令で定義したセグメントのセグメントアドレスが決定される仕組みを解説しましょう．実はアセンブルの時点では各セグメントのセグメントアドレスは決定されません．図4-37-aのように，物理アドレス00000_Hから始まるメモリにプログラムをロードするとした場合のアドレスが割り当てられています．これは「仮のアドレス」にすぎません．

MS-DOSがEXEモデルのプログラムをメモリにロードする際には，そのときのセグメントの割り当て状況に従って，空き領域の先頭から始まるメモリにうまくセグメントを割り当てていきます．このことを示したのが図4-37-bです．

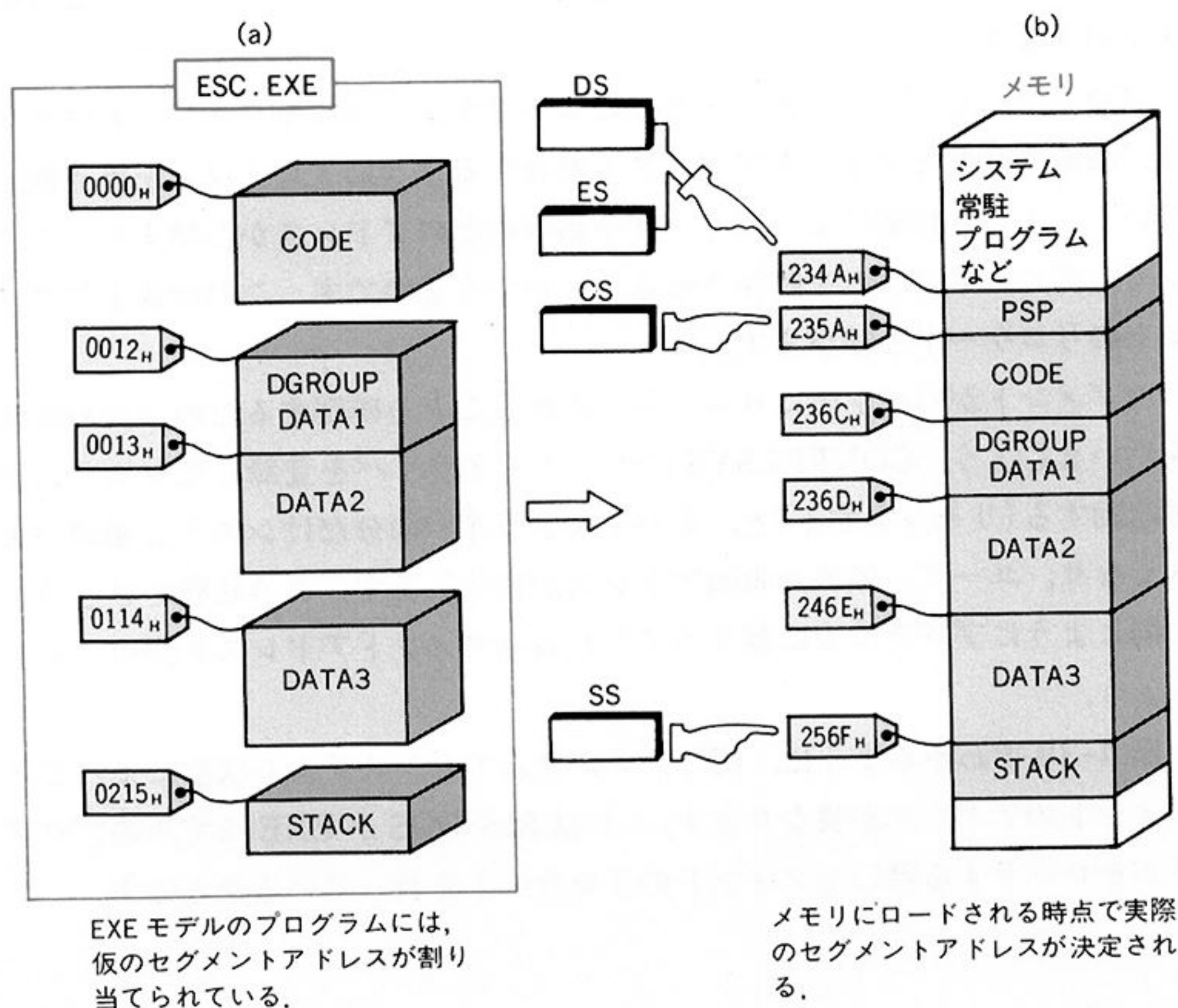



図4-37 EXEモデルロード時のセグメント割り当て

なぜアセンブルの時点でセグメントアドレスを決定できないかというと，プログラムがロードされるアドレスはアセンブル時にはわからないからです．プログラムは4.1節で解説したユーザー領域のメモリにロードされますが，ユーザー領域の先頭アドレスはMS-DOSのバージョンやCONFIG.SYSに登録したバッファの数，デバイスドライバの種類によって異なります．もしもアセンブルの時点でプログラムの実行されるセグメントのアドレスを決めてしまうとすると，プログラムはそのアドレスでしか実行できず，ユーザー領域の先頭アドレスの変化に対応できないことになります．

そこでMS-DOSでは，プログラムをディスクからメモリにロードする時点でセグメントアドレスを決定する仕組みになっています．プログラム中にセグメントアドレスを必要とする命令，たとえばセグメントアドレスをセグメントレジスタにロードするための命令などがあると，そのマシン語命令のオペランド部分をロードされる時点で決まったセグメントアドレスで置き換えていきます．

プログラムをディスクからメモリにロードするときにセグメントを割り当て，決定されたセグメントアドレスと整合するようにプログラムを書き換えていく，という作業によってユーザー領域がどのアドレスから始まるシステムでも同じプログラムを動作させることができるのです．この作業を**セグメントのリロケート**と呼びます．

セグメントがロード時にリロケートされることを確認するために次の実験を行いましょう．CONFIG.SYSにデバイスドライバを登録してシステムを再起動する（リセットする）と，デバイスドライバの分だけシステム領域が大きくなり，ユーザー領域の先頭アドレスが変化します．この状態でさきほどと同じようにプログラムに割り当てられるセグメントアドレスを調べてみましょう．

図4-38でわかるように，同じプログラムでもシステムの状態によってセグメントのアドレスが異なります．これはMS-DOSがEXEモデルのプログラムをロードする際にセグメントのリロケートを行っているからです．

図4-38 セグメントのリロケート

4.8

EXE ファイルの構造と仕組み

EXE ファイルの構造

EXE モデルのプログラムをうまくリロケートできるのは，EXE ファイルにはマシン語プログラム以外にセグメントに関する情報が含まれているからです．この情報は EXE ヘッダと呼ばれ，EXE ファイルの先頭に付いています(図 4-39)．

EXE ヘッダは，プログラムの実行環境を指定する情報とリロケーションのための情報の大きく 2 つに分けられます．前者には，プログラムの実行開始アドレスやスタックとして用意されている領域の大きさ，プログラムの必要とするメモリ量などの情報が収められています．これらの情報は，EXEMOD コマンドによって確認や変更が行えます*．

リロケーション情報は，このプログラムのなかで仮のセグメントアドレスの値がどこで使用されているかといった情報が並んだものです．プログラムをロードしながら，この情報にしたがって，その部分のデータを実際にロードされたアドレス値に変更することでリロケートを実現します．

---

* EXEMOD コマンドは，MASM Ver 4.00，MS-C Ver3.00 以降から提供されている．詳細は各マニュアルを参照のこと．

```
A>REN ESC.EXE ESC.BIN
A>SYMDEB ESC.BIN
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-D 100 3FF
4204:0100  4D 5A 44 01 12 00 02 00-20 00 11 00 FF FF 15 02   MZD.............
4204:0110  00 01 FA 92 00 00 00 00-1E 00 00 00 01 00 01 00   ..z.............
4204:0120  00 00 06 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4204:0130  00 00 00 00 00                      00 00 00   ................
4204:0140                                      00 00 00   ................
4204:0150                 00 00 00-00 00 00 00 00 00 00 00   ................
4204:0160  00 00 00 00 00                00 00 00 00 00   ................
4204:0170  00 00 00 00 00                00 00 00 00 00   ................
4204:0180  00 00 00 00 00                00 00 00 00 00   ................
4204:0190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4204:01A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4204:01B0                 00 00 00-00 00 00 00 00 00 00 00   ................
                                    00 00 00 00 00 00 00   ................
    :02A0  00 00 00 00 00 00
4204:02B0  00 00 00 00 00 00 00 00-00                00 00 00 00   ........
4204:02C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4204:02D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4204:02E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4204:02F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
4204:0300  B8 12 00 8E D8 B8 14 01-8E C0 C7 06 10 00 00 00   ;...[;...CG.....
4204:0310  26 C7 06 00 00 00                02 00 04 00 83 3E   &G.....&G......>
4204:0320  10 00 00 74 12 8B                05 FF 06 12 00 FF   ...t..>.........
4204:0330  0E 10 00 F8 EB 23 90 B4-3F BB 00 00 B9 00 10 BA   ...xk#.4?;..9..:
4204:0340  14 00 CD 21 72 13 0B C0-F9 74 0E 48 A3 10 00 A0   ..M!r..@yt.H#..
                                 -F8 73 03 E9 9D 00 80 3E   ..G.....xs.i...>
    :03B0  00 26 88 07 26 FF                   00 01 90 BB   ...t.<[u"F
4204:03C0  C3 8E DB B4 40 BB 01 00-26                        ....
4204:03D0  CD 21 1F 5B 72 1C 26 3B-06 00 00 F9 75 14         M!.[r.&;...yu.&.
4204:03E0  1E 04 00 26 C7 06 00 00-01 00 26 C7 06 02 00 05   ...&G.....&G....
4204:03F0  00 F8 59 5B 72 22 E2 9C-E9 23 FF 26 83 3E 00 00   .xY[r"b.i#.&.>..
-U 300
4204:0300 B81200         MOV   AX,0012
4204:0303 8ED8           MOV   DS,AX
4204:0305 B81401         MOV   AX,0114
4204:0308 8EC0           MOV   ES,AX
4204:030A C70610000000   MOV   Word Ptr [0010],0000
4204:0310 26C70600000000 MOV   Word Ptr ES:[0000],0000
4204:0317 26C70602000400 MOV   Word Ptr ES:[0002],0004
4204:031E 833E100000     CMP   Word Ptr [0010],+00
-
```

ファイル拡張子が.EXEのままだとリロケートされ，ヘッダが失われてしまうのでリネームする．

EXE型以外のファイルはすべてCOMファイルと同様にオフセットアドレス100_Hからのメモリにロードされる．

ファイルの先頭をダンプする

EXEヘッダ

SSレジスタの初期値

プログラムの開始アドレス(CS:IP)

SPレジスタの初期値

リロケーション情報

セグメントのリロケートを行うための情報

プログラム

プログラムの先頭部分を逆アセンブルしてみる

セグメントDGROUP

セグメントDATA3

に仮のセグメントアドレスが割り当てられている．

EXEファイルのリロケート

EXEモデルのプログラムをロードする際にはこれらの値も実際にロードされたアドレスに変更される

図4-39 EXEファイルの構造（EXEヘッダ）

COMモデルとEXEモデルの違い

EXEモデルのプログラムはアドレスに依存することなく複数のセグメントを扱うことができ，8086CPUの能力をフルに利用することができます．反面，EXEヘッダの分だけ実行ファイルは大きくなり，ロード時にはリロケート処理も行わなければならないので起動には時間がかかります．

これに対し，3章ですでに説明したようにCOMモデルのプログラムにはセグメントが1つしかないので，1つのセグメントを割り当ててそこへロードするだけです．しかも，セグメントが1つしかないということはプログラムのなかでセグメントアドレスを変更することがありません(したがってCOMモデルのプログラムはリロケートする必要がない)．このためEXE

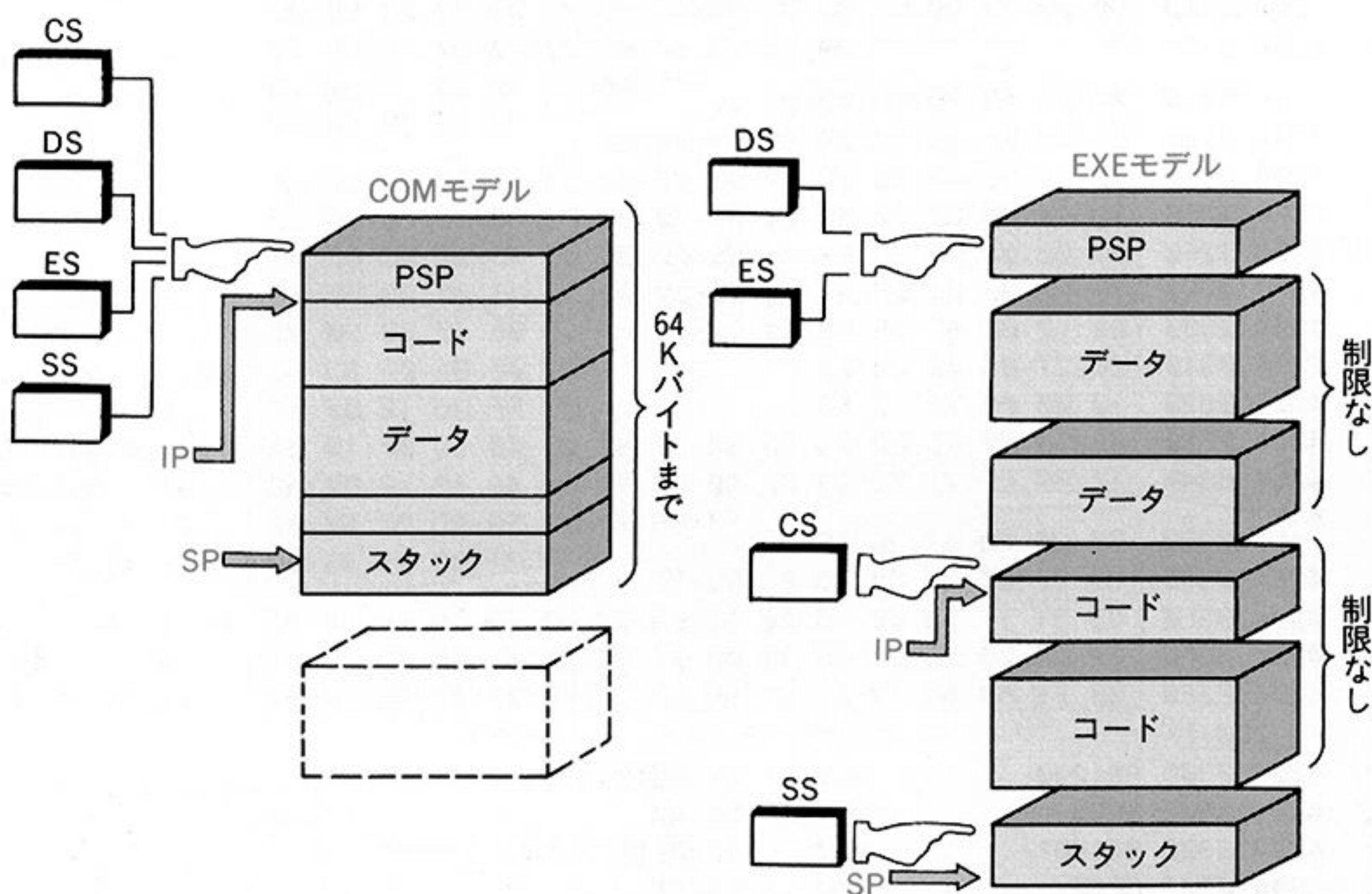



図4-40　COMモデルとEXEモデルの違い

ヘッダのようなものはなく，実行ファイルはプログラムそのものだけからなります．余計なものがないため大きさが小さくなり，ロードも高速です(図4-40)．

COMモデルのプログラムを作成する際には，リンクによって生成されるEXEファイルをEXE2BINコマンドによってCOMファイルに変換します．実はこの作業はEXEファイルから余計なEXEヘッダをとり除く作業です．LINKコマンドはCOMモデルとして書かれたプログラムだろうがとにかくEXEヘッダのついたEXEファイルとして出力しますから，EXE2BINコマンドでEXEヘッダをとり除くことによって初めてCOMモデルのプログラムの実行可能ファイルができるのです．このため実行開始アドレスやスタックとして確保した領域の情報なども失われてしまいます．

COMモデルのプログラムの実行開始アドレスが0100_Hに固定されているわけもこれでおわかりでしょう．

PSP

最後に，たびたび登場したPSPについて説明しておきます．PSPは「Program Segment Prefix」の略であり，MS-DOSがプログラムをロードし実行する際に割り当てるセグメントです．PSPの大きさは100_H(256)バイトであり，MS-DOSからプログラムに渡されるさまざまな情報が格納されています．COMモデルでは実行開始アドレスを0100_Hにすることにより，PSPとプログラムを同じ1つのセグメントに置いています．

PSPにセットされている情報のなかで，最も重要なのは図4-41に示す2つです．1つはプログラムを起動したときにパラメータとして何を指定したかを示すコマンドライン情報です．ここにはコマンドラインに指定したコマンド文字列から，コマンド名を除いた部分が格納されます*．

もう1つは環境セグメントのアドレスです．SETコマンドでセットする環境変数は環境セグメントというセグメントのオフセットアドレス0000_Hから

*なお，リダイレクトやパイプラインを指定する文字列は除かれる．これらはCOMMAND.COMが必要な処理を行うので，呼び出されたプログラムの側では知る必要がないからである．

の領域に格納されて，起動されたプログラムに渡されます．PSP にはそのセグメントのアドレスが格納されます．プログラムはそのアドレスをセグメントレジスタにセットすることにより環境変数を調べることができるようになります*．

SYMDEBは，
A>FIND "Version"README.DOC↵
というコマンドラインが指定された状態をシミュレートする．コマンドラインのうちコマンド名を除いた部分が，PSPに格納される．すなわち
␣"Version"␣README.DOC
という文字列が格納される．

```
E>SYMDEB FIND.EXE "Version" README.DOC ↵
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-D 0 FF↵
1AAF:0000  CD 20 00 A0 00 9A F0 FE-1D F0 28 09 CD 11 C5 09  M . ..p~.p(.M.E.
1AAF:0010  CD 11 F0 08 CD 11 BD 11-03 04 01 00 02 FF FF FF  M.p.M.=.........
1AAF:0020  FF FF FF FF FF FF FF FF-FF FF FF FF AA 1A A2 8D  ............*.".
1AAF:0030  CD 11 14 00 18 00 AF 1A-FF FF FF FF 00 00 00 00  M...../.........
1AAF:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
1AAF:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20  M!K.............
1AAF:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 52 45 41  .............REA
1AAF:0070  44 4D 45 20 20 44 4F 43-00 00 00 00 00 00 00 00  DME  DOC........
1AAF:0080  15 20 22 56 65 72 73 69-6F 6E 22 20 52 45 41 44  . "Version" READ
1AAF:0090  4D 45 2E 44 4F 43 0D 00-0D 4D 45 2E 44 4F 43 0D  ME.DOC...ME.DOC.
1AAF:00A0  00 00 00 00 00 00 00 00-00 00 パラメータの文字列 00  ................
1AAF:00B0  00 スペースの文字コード 00 パラメータの終わりを示すCRコード 00  ................
1AAF:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
1AAF:00D0 パラメータの文字数を示す(15H文字)-00 00 00 00 00 00 00 00  ................
1AAF:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
1AAF:00F0  00 00 00 コマンドラインのパラメータが格納される. 00 00 00  ................
-D 1AAA:0↵
1AAA:0000  43 4F 4D 53 50 45 43 3D-5C 43 4F 4D 4D 41 4E 44  COMSPEC=¥COMMAND
1AAA:0010  2E 43 4F 4D 00 50 41 54-48 3D 41 3A 5C 3B 41 3A  .COM.PATH=A:¥;A:
1AAA:0020  5C 42 49 4E 00 00 01 00-46 49 4E 44 2E 45 58 45  ¥BIN....FIND.EXE
1AAA:0030  00 10 14 00 18 00 AA 1A-FF FF FF FF 00 00 00 00  ......*.........
1AAA:0040  5A AF 1A 51 85 00 00 00-00 00 00 00 00 00 00 00  Z/.Q............
1AAA:0050  CD 20 00 A0 00 9A F0 FE-1D F0 28 09 CD 11 C5 09  M . ..p~.p(.M.E.
1AAA:0060  CD 11 F0 08 CD 11 BD 11-03 04 01 00 02 FF FF FF  M.p.M.=.........
1AAA:0070  FF FF FF FF FF FF FF FF-FF FF FF FF AA 1A A2 8D  ............*.".
-
```

環境変数がセットされているセグメントアドレス

PSPの全内容

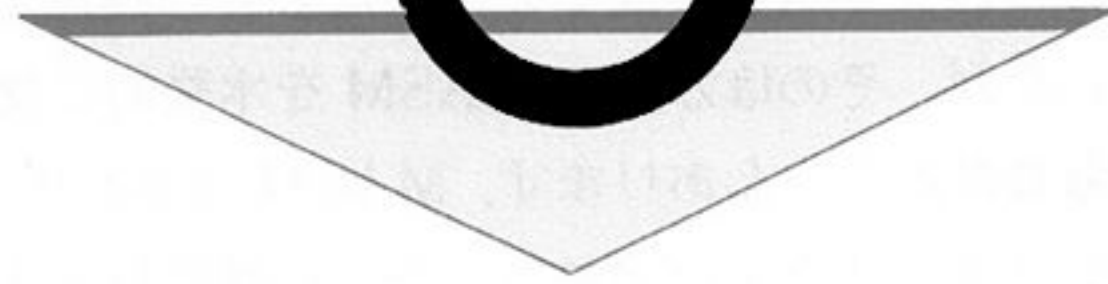
環境文字列領域

1つの環境文字列の終わりを示す00_H

環境文字列のすべての終わりを示す00_H, 00_H

図 4-41　PSP の内容

＊環境セグメントはプログラムを起動するたびに COMMAND.COM の持つ環境セグメントのコピーとして作成される．したがってこの領域の内容を変更してもプログラムが終了するとそのデータは失われ，COMMAND.COM の環境変数には影響を与えない．

5

マクロアセンブラと
モジュール別プログラミング

アセンブラの役割は，マシン語プログラムの開発に役立つ各種の機能を提供することです．これまで解説したように，ラベルやDB擬似命令などはマシン語プログラムの作成に欠かせません．3章ではCOMモデル，4章ではEXEモデルのプログラムを作成するために必要な擬似命令を解説しましたが，そのほかにもMASMを本格的に使うための擬似命令がたくさんあります．MASMは単なる「アセンブラ」ではなく「マクロアセンブラ」と呼ばれるように，大規模なプログラミングをも可能にする便利な機能が豊富に用意されているのです．

本章ではMASMをよりいっそう使いこなすために，こういった機能について解説していきます．

5.1 プログラミングを効率化する擬似命令

本節では3章／4章で取り上げなかった擬似命令のうち，非常に便利な機能を持ったものを解説します．これらの擬似命令を利用することにより，読みやすいプログラムを効率よく開発することができます．

また，本章ではこういった擬似命令を利用して，4章で作成したプログラムESC.ASMを改良していきます．ESC.ASMは，フィルタ型のプログラムの雛形(ひながた)として利用することができますが，これをもとに改良することにより，さまざまテキスト処理プログラムを作ることができるでしょう．本章では変換処理部分のみを変更することにより，ローマ字カナ変換プログラムを作成します．

EQU擬似命令

［書式］ 名前 EQU 定数

MASMではアドレスにラベルという名前を付け，その名前を使ってアドレスを表します．同様に，セグメントにも名前を付けて，その名前で表します．さらにEQU擬似命令を使って，定数値にも名前を付けることができます*．

プログラムのなかではいろいろな定数が必要になりますが，これらを名前で表せると非常に便利です．たとえば例題のプログラムでは，入力用のバッファサイズがたびたび登場します．次ページの図5-1ではこの定数に「BUFSIZ」という名前を付けています．こうすることにより，定数定義の変更だけですべてのバッファサイズを一度に変更することができます．

＊EQU擬似命令には定数定義以外の機能もある．くわしくは5.3節で解説する．

また，MS-DOS のファンクションコールは入出力を行う上で欠かせないものですが，番号で機能を指定しなければならない点が不便です．番号の代わりに名前で機能を指定すれば，よりわかりやすくなるでしょう．図 5-1 のようにファンクション番号に「FC_機能名」のような名前を付けると，番号ではなく自分の付けた名前でファンクションを指定することができるのです．

このほかにも図 5-1 ではいくつかの定数に名前を付けているので，参考にしてください．

```
CR              EQU     ØDH
LF              EQU     ØAH
FC_PUTMSG       EQU     Ø9H
FC_READ         EQU     3FH
FC_WRITE        EQU     4ØH
FC_END          EQU     4CH
STACKSIZ        EQU     1ØØH
BUFSIZ          EQU     1ØØØH

DGROUP  GROUP   DATA1,DATA2
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DGROUP,ES:DATA3,SS:STACK

        MOV     ES:OUT_P
        CLC
PUTC_END:
;-- put char end --
        POP     CX
        POP     BX
        JC      QUIT
        LOOP    PUTS
PUTS_END:
        JMP     M_LOOP
;
FLUSH:
;-- remain output buffer ? --
        CMP     WORD PTR ES:OUT_LEN,Ø
        JE      QUIT
;-- buffer flush --
        PUSH    DS
        MOV     BX,ES
        MOV     DS,BX
        MOV     AH,FC_WRITE
        MOV     BX,1
        MOV     CX,WORD PTR ES:OUT_LEN
        MOV     DX,OFFSET ES:OUT_BUF
        INT     21H
        POP     DS
```

EQU擬似命令を用いて，定数値に名前を定義する

プログラム中で定数名を使うと，アセンブル時にはその定数名に定義された値として解釈される

```
;-- buffer flush end --
QUIT:
        MOV     AH,FC_END
        INT     21H
        DB      'K','カ','キ',...
        DB      'S','サ','シ','ス','セ',...
        DB      'T','タ','チ','ツ','テ','ト'
        DB      'N','ナ','ニ','ヌ','ネ','ノ'
        DB      'H','ハ','ヒ','フ','ヘ','ホ'
        DB      'M','マ','ミ','ム','メ','モ'
        DB      'Y','ヤ','イ','ユ','エ','ヨ'
        DB      'R','ラ','リ','ル','レ','ロ'
        DB      'W','ワ','イ','ウ','エ','ヲ'
;
DATA1   ENDS

DATA2   SEGMENT
;
IN_LEN  DW      ?
IN_PTR  DW      ?
IN_BUF  DB      BUFSIZ DUP (?)
;
DATA2   ENDS

DATA3   SEGMENT
;
OUT_LEN DW      ?
OUT_PTR DW      ?
OUT_BUF DB      BUFSIZ DUP (?)
;
DATA3   ENDS

STACK   SEGMENT STACK
        DB      STACKSIZ DUP (?)
STACK   ENDS

        END START
```

図 5-1　EQU 擬似命令による定数の置換

定数に名前を付けることには，次のようなメリットがあります．これにより，プログラムが非常に書きやすくなることが理解できるでしょう．

・定数の名前から意味がすぐにわかり，プログラムをあとで読み返したときにもわかりやすくなる．
・定数のタイプミスなどによる間違いが減少する．
・定数に変更があったときに，1箇所を変更するだけでよい．

INCLUDE 擬似命令

［書式］ INCLUDE ファイル名

前項で解説したように，EQU 擬似命令を使って MS-DOS のファンクションコールの番号を定義しておけば，いちいち番号を調べなくても名前で指定すればよいのでたいへん便利です．そうすると，この定義はいろいろなプログラムで利用したくなります．この定義を他のプログラムで利用するにはどうすればよいでしょうか．

1つにはエディタの機能を使って，他のソースファイルから別のソースファイルへ定義の部分をコピーする方法があります．これは実際によく使われる方法です．しかし，この方法には欠点があります．定義に間違いがあった場合，追加や変更を加える際に，コピーしたソースファイル全部に同じ変更を加えなければなりません．

もっとも簡単でうまい方法は，INCLUDE 擬似命令を使う方法です．INCLUDE 擬似命令の使用例を図 5-2 に示します．

ヘッダファイル MSDOS.H

```
CR              EQU     0DH
LF              EQU     0AH
FC_PUTMSG       EQU     09H
FC_READ         EQU     3FH
FC_WRITE        EQU     40H
FC_END          EQU     4CH
```

INCLUDE擬似命令により，ヘッダファイルの内容が，この部分にあるかのように扱われる．

ソースファイル ESC.ASM

```
        INCLUDE MSDOS.H

STACKSIZ        EQU     100H
BUFSIZ          EQU     1000H

DGROUP  GROUP   DATA1,DATA2
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DGROUP,ES:DATA3,SS:STACK
```

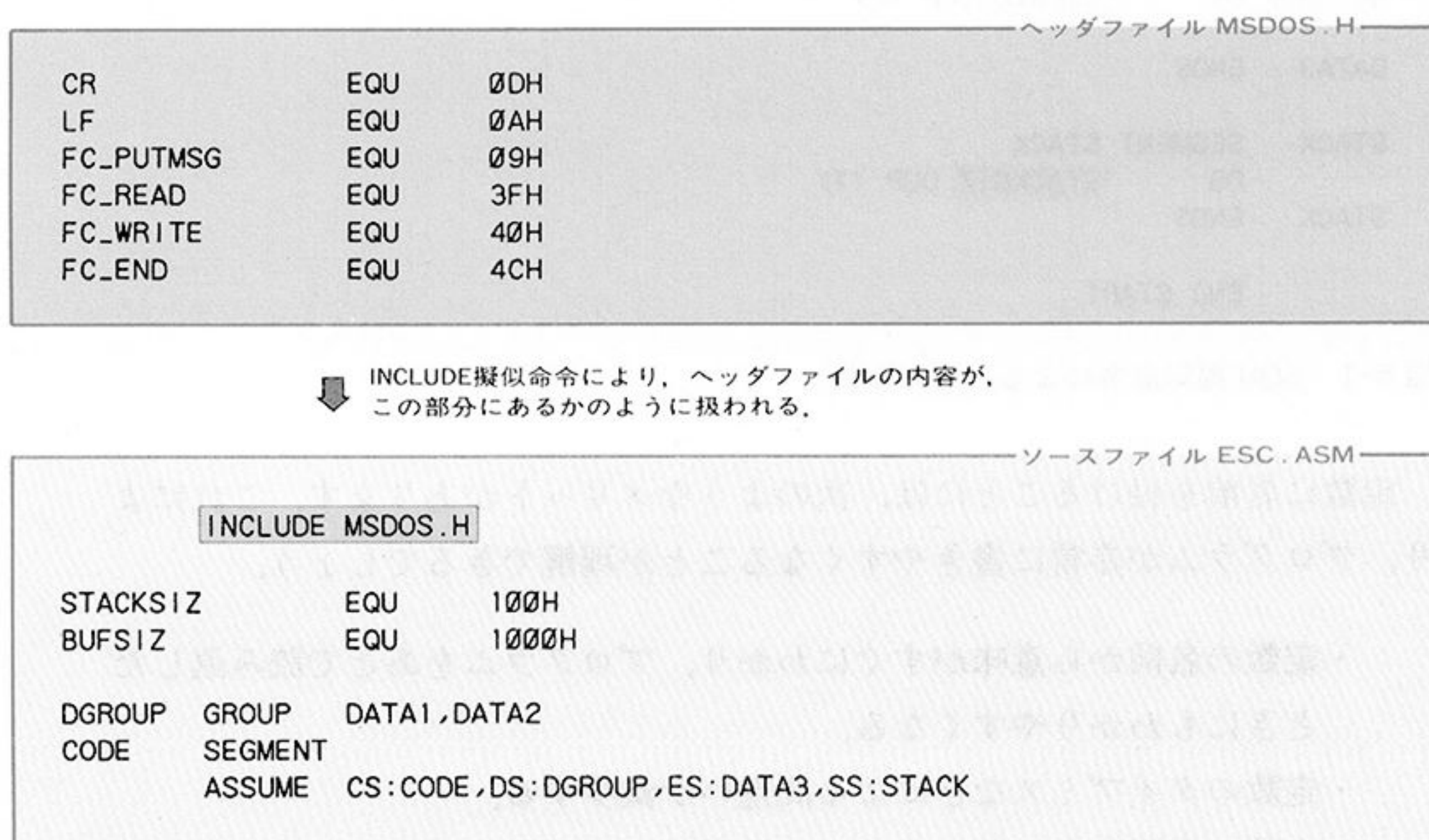



図 5-2　INCLUDE 擬似命令の使用例

まず，定数定義の部分のみを書いたファイルを用意します．このようなファイルは，プログラムの頭の方で書くべき内容なので，**ヘッダファイル**と呼びます．そして，INCLUDE擬似命令でそのヘッダファイルの名前を指定します．

INCLUDE(インクルード)には「含む」という意味があり，指定したファイルの内容がそこに書かれているかのように扱えます．つまり，別のファイルの内容をそこへコピーしたのと同じ効果があるのです．

INCLUDE擬似命令によって，先に挙げた問題点は解決されます．定数定義だけを記述したファイルを用意し，その定義を利用するプログラムからINCLUDE擬似命令によってファイルを取り込めばよいのです．どのプログラムからでも，同じファイルをインクルードすることによりまったく同じ定義を利用することができます．さらに，定義に変更や追加があった場合にも定義ファイルだけを修正すれば，プログラムのソースファイルに手を入れる必要はありません．

PROC～ENDP擬似命令

［書式］　プロシージャ名　PROC　［属性］
　　　　　　　⋮
　　　　　プロシージャ名　ENDP

・プロシージャ名は｛アルファベット，@，$，_，?，数字｝からなる文字列で，数字で始まることはできない．
・属性はNEARまたはFAR．省略すると，NEARを指定したことになる．

PROC擬似命令はPROCedure(プロシージャ：手続き)の略であり，プロシージャを定義する擬似命令です．プロシージャは簡単にいえばサブルーチンのことですが，MASMではサブルーチンに名前を付け，構造化したものを指します．プログラム中の1つのまとまった処理をプロシージャとしてプログラム本体とは別に定義することで，プログラムの構造をはっきりさせることができます．

4章の例題プログラムをもとに作成したローマ字カナ変換プログラム

「ROMA.ASM」を以下のリスト5-1に示します．入出力のバッファリング処理，変換処理をプロシージャとして定義しているため，プログラム本体の構造がかなりわかりやすくなっています．

リスト5-1　ローマ字カナ変換プログラム ROMA.ASM

```
        INCLUDE MSDOS.H

STACKSIZ        EQU     100H
BUFSIZ          EQU     1000H

DGROUP  GROUP   DATA1,DATA2
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DGROUP,ES:DATA3,SS:STACK
                                 メインルーチン
START:
        MOV     AX,DGROUP        4章のリスト4-2と較べてみると，処理の単位を
        MOV     DS,AX            プロシージャにすることで，プログラムの構造
        MOV     AX,DATA3         がかなりわかりやすくなっている
        MOV     ES,AX
;-- init buffer for read/write --
        MOV     WORD PTR IN_LEN,0
        MOV     WORD PTR ES:OUT_LEN,0
        MOV     ES:OUT_PTR,OFFSET ES:OUT_BUF
M_LOOP:
        CALL    GETC        ……1文字入力プロシージャを呼び出す
        JC      FLUSH
        CALL    ROMAKANA    ……ローマ字カナ変換プロシージャを呼び出す
PUTS:
        CMP     CX,0
        JE      PUTS_END
        MOV     AL,[BX]
        INC     BX
        PUSH    BX
        PUSH    CX
        CALL    PUTC        ……1文字出力プロシージャを呼び出す
        POP     CX
        POP     BX
        JC      QUIT
        LOOP    PUTS
PUTS_END:
        JMP     M_LOOP
;
;-- remain output buffer ? --
FLUSH:
        CMP     WORD PTR ES:OUT_LEN,0
        JE      QUIT
        CALL    FLUSH_SUB   ……バッファフラッシュプロシージャを呼び出す
QUIT:
        MOV     AH,FC_END
        INT     21H
```

1文字入力プロシージャGETC

```
;-- get char --
GETC    PROC
        CMP     WORD PTR IN_LEN,0
        JE      GETC_1
        MOV     DI,IN_PTR
        MOV     AL,[DI]
        INC     WORD PTR IN_PTR
        DEC     WORD PTR IN_LEN
        CLC
        JMP     GETC_END
GETC_1:
        MOV     AH,FC_READ
        MOV     BX,0
        MOV     CX,BUFSIZ
        MOV     DX,OFFSET DGROUP:IN_BUF
        INT     21H
        JC      GETC_END
        OR      AX,AX
        STC
        JZ      GETC_END
        DEC     AX
        MOV     IN_LEN,AX
        MOV     AL,IN_BUF
        MOV     IN_PTR,OFFSET DGROUP:IN_BUF+1
        CLC
GETC_END:
        RET
GETC    ENDP
```

(このプロシージャは，リスト4-2の1文字入力ルーチンの部分をプロシージャにしたものである)

1文字出力プロシージャPUTC

```
;-- put char --
PUTC    PROC
        CMP     WORD PTR ES:OUT_LEN,BUFSIZ
        JE      PUTC_1
        INC     WORD PTR ES:OUT_LEN
        MOV     BX,ES:OUT_PTR
        MOV     ES:[BX],AL
        INC     WORD PTR ES:OUT_PTR
        CLC
        JMP     PUTC_END
PUTC_1:
        PUSH    AX
        CALL    FLUSH_SUB
        POP     BX
        JC      PUTC_END
        CMP     AX,ES:OUT_LEN
        STC
        JNE     PUTC_END
        MOV     ES:OUT_BUF,BL
        MOV     WORD PTR ES:OUT_LEN,1
        MOV     ES:OUT_PTR,OFFSET ES:OUT_BUF+1
        CLC
```

(このプロシージャは，リスト4-2の1文字出力ルーチンの部分をプロシージャにしたものである)

```
PUTC_END:
        RET
PUTC    ENDP
```

バッファフラッシュプロシージャFLUSH_SUB

```
;-- buffer flush --
FLUSH_SUB       PROC
        PUSH    DS
        MOV     BX,ES
        MOV     DS,BX
        MOV     AH,FC_WRITE
        MOV     BX,1
        MOV     CX,ES:OUT_LEN
        MOV     DX,OFFSET ES:OUT_BUF
        INT     21H
        POP     DS
        RET
FLUSH_SUB       ENDP
```

このプロシージャは，リスト4-2のバッファフラッシュルーチン（バッファに残っているデータをまとめて出力する処理）をプロシージャにしたものである

ローマ字カナ変換プロシージャROMAKANA

```
;-- roma kana convert --
ROMAKANA        PROC
        MOV     WORD PTR CNV_LEN,Ø
        CMP     BYTE PTR CHR1F,Ø     1文字目を変換
        JNE     SECONDCHR            途中かどうか
        CALL    ISALPHA
        JE      FSTCHR               アルファベットでなければ,
        JMP     SET_END              変換はできない
FSTCHR:
        AND     AL,5FH ……………強制的に大文字に変換する
        CALL    BOIN                 母音でなければ,
        JNE     STORE                とりあえず記録する
        ;
        MOV     AL,HYOU2[BX]         母音テーブルから
        JMP     SET_END              変換結果を取り出す
STORE:
        MOV     BYTE PTR CHR1F,1     1文字目をとりあえず保存し,
        MOV     CHR1,AL              変換途中であることを示すフ
        JMP     ROMAKANA_END         ラグをセットする
SECONDCHR:
        CALL    ISALPHA              2文字目がアルファベットで
        JZ      TRANSFER             なければ,変換できない
        ;
        XCHG    AL,CHR1 ……………記録しておいた1文字目と交換する
        ;
        CALL    SETCHR ……………1文字目をそのまま変換後の文字としてセットする
        MOV     AL,CHR1              2文字目も変換後の文字としてセットする
        MOV     BYTE PTR CHR1F,Ø     変換途中であることを示すフラグはクリアする
        JMP     SET_END
TRANSFER:
        AND     AL,5FH ……………強制的に大文字に変換する（「はじめて読む8086」201ページ参照）
        XCHG    AL,CHR1 ……………記録しておいた1文字目と交換する
        ;
        MOV     SI,OFFSET DGROUP:HYOU3
        MOV     CX,9
```

入力パラメータ	：ALレジスタに変換したい文字を格納して呼び出す
処理	：CXレジスタに変換後の文字数，BXレジスタに変換後の文字列のアドレスを格納して返す

注：完全に変換できない文字の場合は，とりあえず文字数を0として返し，次の呼び出しで変換を完成させてから結果を返す．

この2行を
JNE SET_END
としてもよさそうだが，条件ジャンプは－128～＋127バイトの範囲にしかジャンプできないのでこうしている．アセンブル時に，
Relative jump out of range
というエラーが発生したらこのようにする．

```
SLOOP:
        CMP     [SI],AL         子音変換テーブルから，1文字目と
        JE      SFOUND          一致するものを探す
        ADD     SI,6
        LOOP    SLOOP
        ;
        CALL    SETCHR ……………見つからなければ1文字目をそのまま返す
        MOV     AL,CHR1
        CALL    BOIN
        JNE     ROMAKANA_END            2文字目について1文字目
        MOV     AL,HYOU2[BX]            と同じ処理を行う
        MOV     BYTE PTR CHR1F,Ø
        JMP     SET_END
;
SFOUND:
        XCHG    AL,CHR1
        CALL    BOIN            1文字目が子音テーブルから見つかった場合，
        JNE     CHKN            2文字目が母音かどうか調べる
        ;
        MOV     AL,[SI+BX+1]            2文字目が母音ならば，子音
        MOV     BYTE PTR CHR1F,Ø        変換テーブルから変換結果が
        JMP     SET_END                 得られる
CHKN:
        CMP     CHR1,'N'
        JNE     NOTKANA
        CMP     AL,'N'
        JNE     NOTKANA
        ;                               'ン'の処理
        MOV     AL,'ン'
        MOV     BYTE PTR CHR1F,Ø
        JMP     SET_END
NOTKANA:
        XCHG    AL,CHR1         'ン'でもなければ変換不能として
SET_END:                        1文字目をそのまま返し，2文
        CALL    SETCHR          字目を記録する
ROMAKANA_END:
        MOV     CX,CNV_LEN                      変換後の文字列の長さと，変換結果
        MOV     BX,OFFSET DGROUP:CNV_BUF        の文字列を格納したメモリのアドレ
        RET                                     スをレジスタにセットしてリターンする
ROMAKANA        ENDP
```

変換結果格納プロシージャSETCHR

```
;-- set char --
SETCHR  PROC
        MOV     BX,CNV_LEN
        MOV     SI,OFFSET DGROUP:CNV_BUF
        MOV     [SI+BX],AL
        INC     WORD PTR CNV_LEN
        RET
SETCHR  ENDP    (この処理は，プロシージャROMAKANA中で
                 何度も必要なためプロシージャとした)
```

入力パラメータ：ALレジスタに変換後の文字を入れて呼び出す

処理：変換結果を内部バッファに格納し，文字数をインクリメントする

母音判定プロシージャBOIN

```
;-- boin --
BOIN    PROC
        MOV     BX,0
BLOOP_1:
        CMP     HYOU1[BX],AL
        JE      BFOUND_1
        INC     BX
        CMP     BX,5
        JBE     BLOOP_1
BFOUND_1:
        RET
BOIN    ENDP
```

母音テーブルから一致する文字を探す

入力パラメータ：ALレジスタに調べたい文字を入れて呼び出す
処理　　　　　：文字が母音ならばゼロフラグをセットし，対応する番号をBXレジスタに入れて返す

アルファベット判定プロシージャISALPHA

```
;-- isalpha --
ISALPHA PROC
        CMP     AL,'A'
        JB      NOTALPHA_2
        CMP     AL,'Z'
        JBE     ALPHA_2
        CMP     AL,'a'
        JB      NOTALPHA_2
        CMP     AL,'z'
        JA      NOTALPHA_2
ALPHA_2:
        CMP     AL,AL
NOTALPHA_2:
        RET
ISALPHA ENDP
```

文字が'A'～'Z' 'a'～'z'の範囲にあるかどうか調べる

入力パラメータ：ALレジスタに調べたい文字を入れて呼び出す
処理　　　　　：文字がアルファベットならば，ゼロフラグをセットする．

CMP AL,AL ……アルファベットであったときの処理．ALレジスタとALレジスタをCMP命令で比較すると，必ずゼロフラグがセットされる

```
CODE    ENDS

DATA1   SEGMENT
;
CHR1F   DB      0
CHR1    DB      0
CNV_LEN DW      ?
CNV_BUF DB      2 DUP (?)
;       母音番号 0   1   2   3   4
HYOU1   DB      'A','I','U','E','O'
HYOU2   DB      'ア','イ','ウ','エ','オ'
;               0   1   2   3   4   5
HYOU3   DB      'K','カ','キ','ク','ケ','コ'
        DB      'S','サ','シ','ス','セ','ソ'
        DB      'T','タ','チ','ツ','テ','ト'
        DB      'N','ナ','ニ','ヌ','ネ','ノ'
        DB      'H','ハ','ヒ','フ','ヘ','ホ'
        DB      'M','マ','ミ','ム','メ','モ'
        DB      'Y','ヤ','イ','ユ','エ','ヨ'
        DB      'R','ラ','リ','ル','レ','ロ'
        DB      'W','ワ','イ','ウ','エ','ヲ'
;
DATA1   ENDS
```

CHR1F ……1文字目を変換中かどうかを示すフラグ
CHR1 ……1文字目を記録しておくための領域
CNV_LEN ……変換後の文字列の文字数を格納する領域
CNV_BUF ……変換後の文字列を格納する領域

HYOU1/HYOU2 ……母音変換テーブル

HYOU3 ……子音変換テーブル

SIレジスタ………子音のアドレス
BXレジスタ………母音番号
ならば，
[SI＋BX＋1]で変換結果を取り出せる

```
DATA2    SEGMENT
;
IN_LEN   DW       ?
IN_PTR   DW       ?
IN_BUF   DB       BUFSIZ DUP (?)
;
DATA2    ENDS

DATA3    SEGMENT
;
OUT_LEN DW        ?
OUT_PTR DW        ?
OUT_BUF DB        BUFSIZ DUP (?)
;
DATA3    ENDS

STACK    SEGMENT STACK
         DB       STACKSIZ DUP (?)
STACK    ENDS

         END START
```

リスト5-1でわかるようにPROC擬似命令でプロシージャを定義する方法は，SEGMENT擬似命令でセグメントを定義する方法によく似ています．すなわち，

名前　PROC　型属性

⋮

プロシージャ本体

⋮

名前　ENDP

としてPROCとENDPでプロシージャを囲みます．型属性には次の2つがあります．

NEAR　同一セグメントからコールされるプロシージャであることを示す．

FAR　他のセグメントからコールされるプロシージャであることを示す．

複数のコードセグメントを利用するかどうかによって，どちらかの型属性を選択しなければなりません．両者の違いを次の図 5-3 に示します．

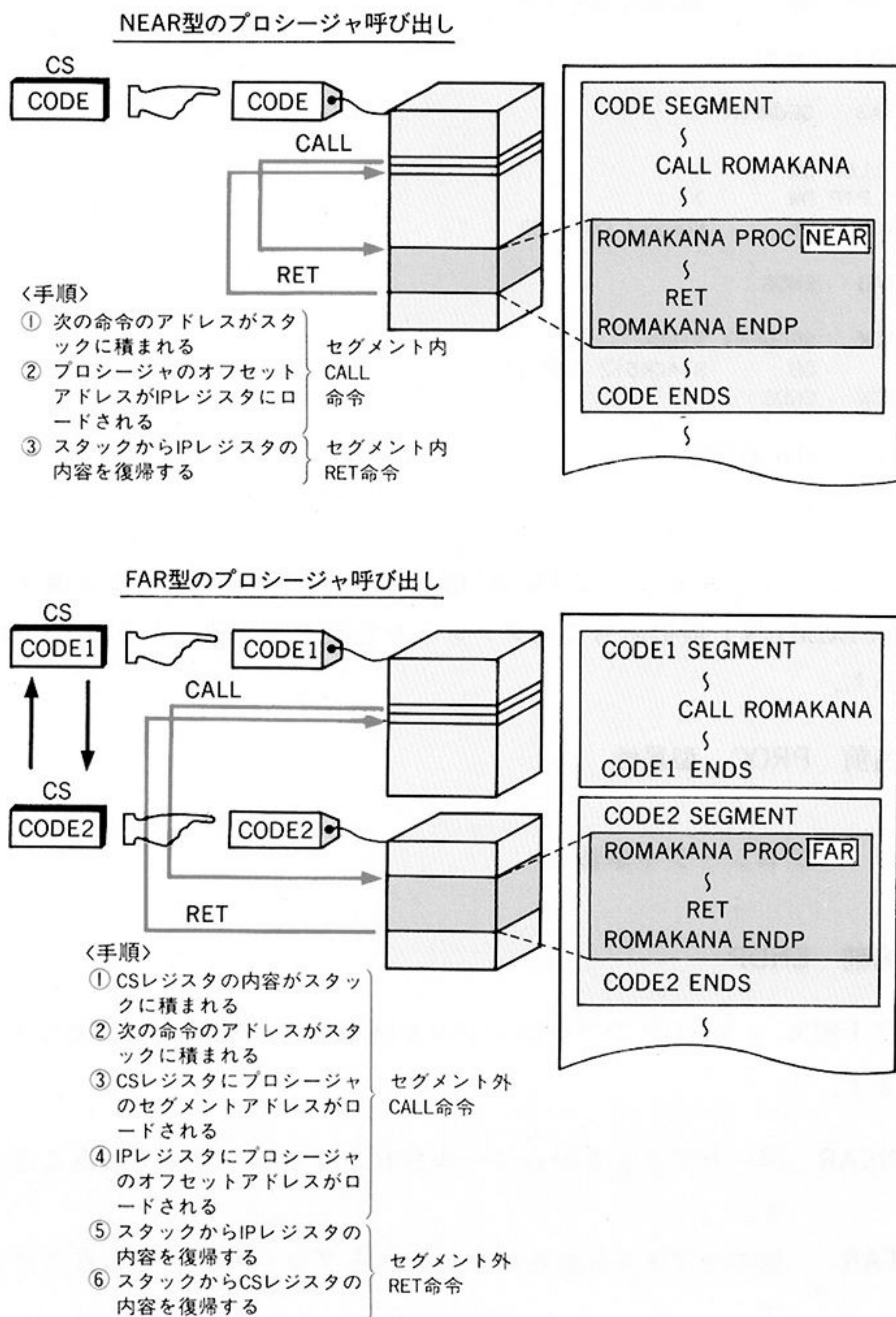



図 5-3　NEAR 型プロシージャと FAR 型プロシージャ

NEAR型とFAR型の違いは，CALL命令およびRET命令がセグメント外とセグメント内の2種類あることによります．MASMはNEAR型のプロシージャではセグメント内CALL／RETのマシン語命令を，FAR型のプロシージャについてはセグメント外CALL／RETのマシン語命令をそれぞれ出力します．

コードセグメントが1つしかなく，同じセグメント内から呼び出されるプロシージャならNEAR型です．これに対し，コードセグメントが複数ありプロシージャの属するセグメント以外からも呼び出されるプロシージャならばFAR型です．また，FAR型のプロシージャは，同じセグメント内から呼び出される場合にもセグメント外CALL命令によって呼び出されます．

型属性を指定しなかった場合には，NEAR型が指定されたことになります．コードセグメントを複数に分けるかどうかによって，プロシージャの型属性を変更しなければならないことに注意してください*.

さて，ROMA.ASMはEXEモデルのプログラムですから，アセンブル&リンクの方法は4章で解説したとおりです．各自で試してみてください．

ローマ字カナ変換プログラムの利用例を図5-4に示します．ここでは，1章で紹介している天気予報プログラムの結果を，MS-DOSのパイプ機能によってカナに変換しています**.

```
A>OTENKI | ROMA ↵  ←――――――何かキーを押す
ｸﾓﾘ ﾉﾁ ｱﾒ ……………半角カナに変換された

A>
```

図5-4 ROMAコマンドの実行例

---

*C言語などの高級言語とアセンブラのプロシージャをリンクする場合には，高級言語側からFAR型でプロシージャを呼び出すのかNEAR型で呼び出すのかを調べて一致させなければならない．一般にラージモデルと呼ばれるメモリモデルではFAR型，スモールモデルと呼ばれるメモリモデルではNEAR型を指定する．

**すべてのローマ字をカナに変換できるわけではない．読者の改良にまかせている．

PTR 演算子(2)

[書式] FAR　PTR　プロシージャ名

プロシージャの呼び出しについて，注意しなければならないことを解説しておきましょう．特に指定しなければ，プロシージャの型属性はNEAR型であると仮定されます．これはプロシージャを呼び出す場合も同様です．したがって，FAR型のプロシージャを前方参照のかたちで呼び出している場合には，型属性の不一致が発生します．呼び出しの時点ではNEAR型のプロシージャとして扱われるのに，その後FAR型で定義されていることになるからです．

このような場合には，プロシージャを呼び出すところでそのプロシージャがFAR型であることをMASMに指示しておかねばなりません．そのために用いられるのがPTR演算子です．3章ではデータの型を指示するためにPTR演算子を使用しましたが，今度はこれをプロシージャの型を指示するために使用します．PTR演算子の使い方を次ページの図5-5に示します．

IF 擬似命令

```
[書式] IF 式
         ∫
       [ELSE]
         ∫
       ENDIF
```

プログラムは一度で思いどおりに動いてくれるとは限りません．最初のうちはうまく動かずに，何度も修正してはアセンブルし実行してみるということを繰り返すものです．特にマシン語のプログラムは，ちょっとしたことで暴走してしまうことも少なくありません．

そこで，どこまで実行してから暴走してしまったのかを調べるために，プログラムの各所でメッセージを出力させる方法があります．プログラムを実行しながら，随所にメッセージを出力すればどこまで確実に実行されたのか

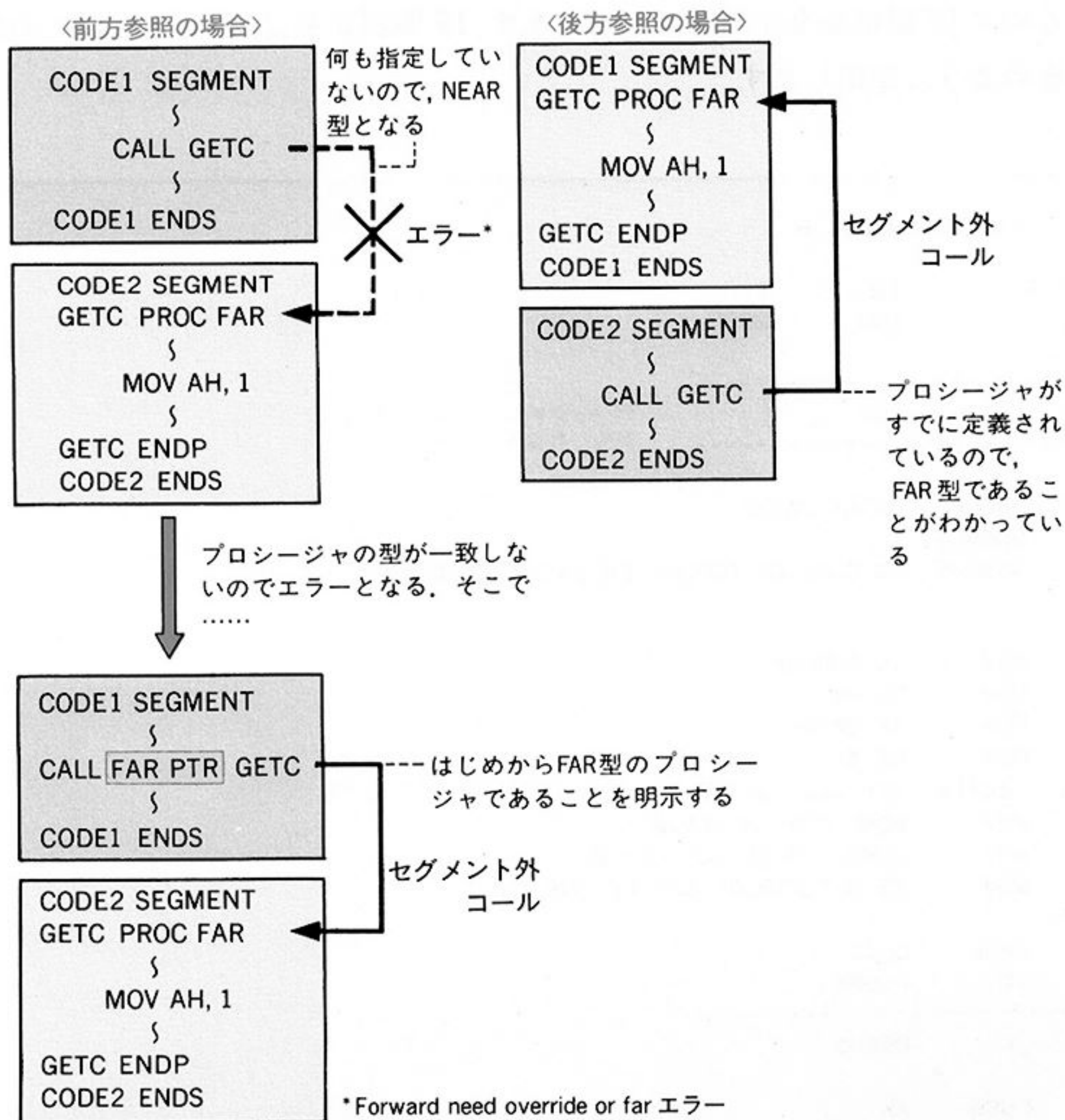



図 5-5 FAR 型プロシージャの前方参照

がわかります. プログラムを実行してみて最後にメッセージが出力された箇所から次にメッセージが出力されるべき箇所までの間で問題が発生しているのです.

こうしてプログラムのミスが発見されると余計なメッセージの出力はもはや必要ありません. 邪魔なのですべて削除してしまいましょう…. でもプログラムの改良を続けていくうちにまた暴走するようになるかもしれません. では残しておきましょう…. いつまで残しておけばよいのでしょう?

このように場合によってプログラムの一部を有効にしたり無効にしたりす

るために IF 擬似命令が用意されています. IF 擬似命令は, たとえば以下の図 5-6 のように使用します.

```
        INCLUDE MSDOS.H

STACKSIZ        EQU     100H
BUFSIZ          EQU     1000H

;--------------------------------
DEBUG           EQU     1     ………デバッグ時には0以外を指定し,
                                  完成したら0を指定する
;--------------------------------

DGROUP  GROUP   DATA1,DATA2
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DGROUP,ES:DATA3,SS:STACK

START:
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     AX,DATA3
        MOV     ES,AX
;-- init buffer for read/write --
        MOV     WORD PTR IN_LEN,0
        MOV     WORD PTR ES:OUT_LEN,0
        MOV     ES:OUT_PTR,OFFSET ES:OUT_BUF
M_LOOP:
        CALL    GETC
        JC      FLUSH
;--------------------------------
        IF      DEBUG
;
        PUSH    AX
        MOV     AL,CR
        CALL    PUTC
        MOV     AL,LF
        CALL    PUTC
        POP     AX    ………この領域は, DEBUGの値が0以外のときにアセンブル
        PUSH    AX        されるが, 0のときにはアセンブルされない
        CALL    PUTC
        MOV     AL,'-'
        CALL    PUTC
        POP     AX
;
        ENDIF
;--------------------------------
        CALL    ROMAKANA
PUTS:
        CMP     CX,0
        JE      PUTS_END
```

図 5-6 IF 擬似命令の使用例

IF擬似命令では，式によって条件を指定します．アセンブラはその式が「0以外」であればENDIFまでのプログラムをアセンブルし，「0」であればアセンブルしません．ですから，DEBUGという定数の定義を0にすれば，デバッグのためにメッセージは出力されなくなるわけです．再びメッセージが必要になれば，DEBUGの定義を1にしてアセンブルしなおせばよいのです．

式の値が0以外ならば「真」，0ならば「偽」というわけです．ELSE擬似命令を使って条件が偽の場合のプログラムを書いておくこともできます．この例を図5-7に示します．

```
        INCLUDE MSDOS.H

STACKSIZ        EQU     100H
BUFSIZ          EQU     1000H

SMALL           EQU     0       } SMALLのとき0, LARGEのとき1と定義
LARGE           EQU     1       }
MODEL           EQU     SMALL   ……MODELはSMALLである


DGROUP  GROUP   DATA1,DATA2
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DGROUP,ES:DATA3,SS:STACK
        MOV     AH,FC_WRITE
        MOV     BX,1
        MOV     CX,ES:OUT_LEN
        MOV     DX,OFFSET ES:OUT_BUF
        INT     21H
        POP     DS
        RET
FLUSH_SUB       ENDP

;-- roma kana convert --
        IF      MODEL
ROMAKANA        PROC    FAR     ……MODELの値がLARGE(0以外)のときに
                                  アセンブルされる部分
        ELSE
ROMAKANA        PROC            ……MODELの値がSMALL(0)のときに
                                  アセンブルされる部分
        ENDIF
        MOV     WORD PTR CNV_LEN,0
        CMP     BYTE PTR CHR1F,0
        JNE     SECONDCHR
        CALL    ISALPHA
        JE      FSTCHR
        JMP     SET_END
```

図5-7 IF～ELSE～ENDIF擬似命令の使用例

この例は先のPROC擬似命令で，メモリモデルの違いによってプロシージャの属性を変更する操作を簡単にするためのものです．LARGEという定数を真(1)，SMALLという定数を偽(0)に定義してあるので，MODELという定数をそのどちらに定義するかによって自動的にプロシージャの属性を変更することができます．

この方法ならば，モジュール内に複数のプロシージャがある場合にも，1つの定数MODELを変更するだけでメモリモデルの変更に対応することができます．

MASMのIF擬似命令は，高級言語におけるifなどの条件分岐命令とは意味が異なるので注意が必要です．高級言語のifは，プログラム実行時に式の値を調べて分岐します．したがって条件が満たされるかどうかは実行時に決まり，どちらの条件のコードも生成されています．ところがMASMの場合にはアセンブルの時点で式の値を調べ，あてはまる条件の部分しかアセンブルを行いません．条件に合わない部分は実行ファイルには含まれず，最初からなかったかのように扱われるのです．

表5-1に，条件アセンブル擬似命令の一覧表を示しておきます．

条件アセンブル擬似命令	条件ブロックがアセンブルされる場合
IF 式	<式>が0以外の値に評価された場合
IFE 式	<式>が0に評価された場合
IF1	アセンブラがパス1を実行中の場合
IF2	アセンブラがパス2を実行中の場合
IFDEF <名前>	<名前>が定義されているか，EXTRN擬似命令により外部参照として宣言されている場合
IFNDEF <名前>	<名前>が定義されておらず，外部参照としても宣言されていない場合
IFB <[引数]>*	<引数>がない場合
IFNB <[引数]>*	<引数>がある場合
IFIDN <引数1>，<引数2>	文字列<引数1>と文字列<引数2>が同一である場合
IFDIF <引数1>，<引数2>	文字列<引数1>と文字列<引数2>が異なる場合

*[ ]は省略可能

表5-1 条件アセンブル擬似命令

5.2 マクロアセンブラとは

本節ではマクロアセンブラが，単なるアセンブラ以上にすぐれた機能を持ったものであることを解説します．アセンブラの役割を機能ごとに分解して，「マクロ」部分の役割を明らかにするために，アセンブラの機能を段階的に追っていきましょう．

ハンドアセンブル

アセンブルのもっとも原始的な形は，いわゆる「ハンドアセンブル」でしょう．ハンドアセンブルとは手でアセンブルする，つまり人間が自分でアセンブルすることです．次ページの図 5-8 のようなアセンブリ言語のニーモニックとマシン語コードの対応表を見ながら，アセンブリ言語で書いたソースプログラムをオブジェクトプログラムに変換するのです．

これはやってみればわかりますが，実にたいへんな作業であり非人間的な作業です．しかし，マイクロコンピュータを初期に勉強した人はアセンブラを利用することもできず，苦労してハンドアセンブルをしたのです．

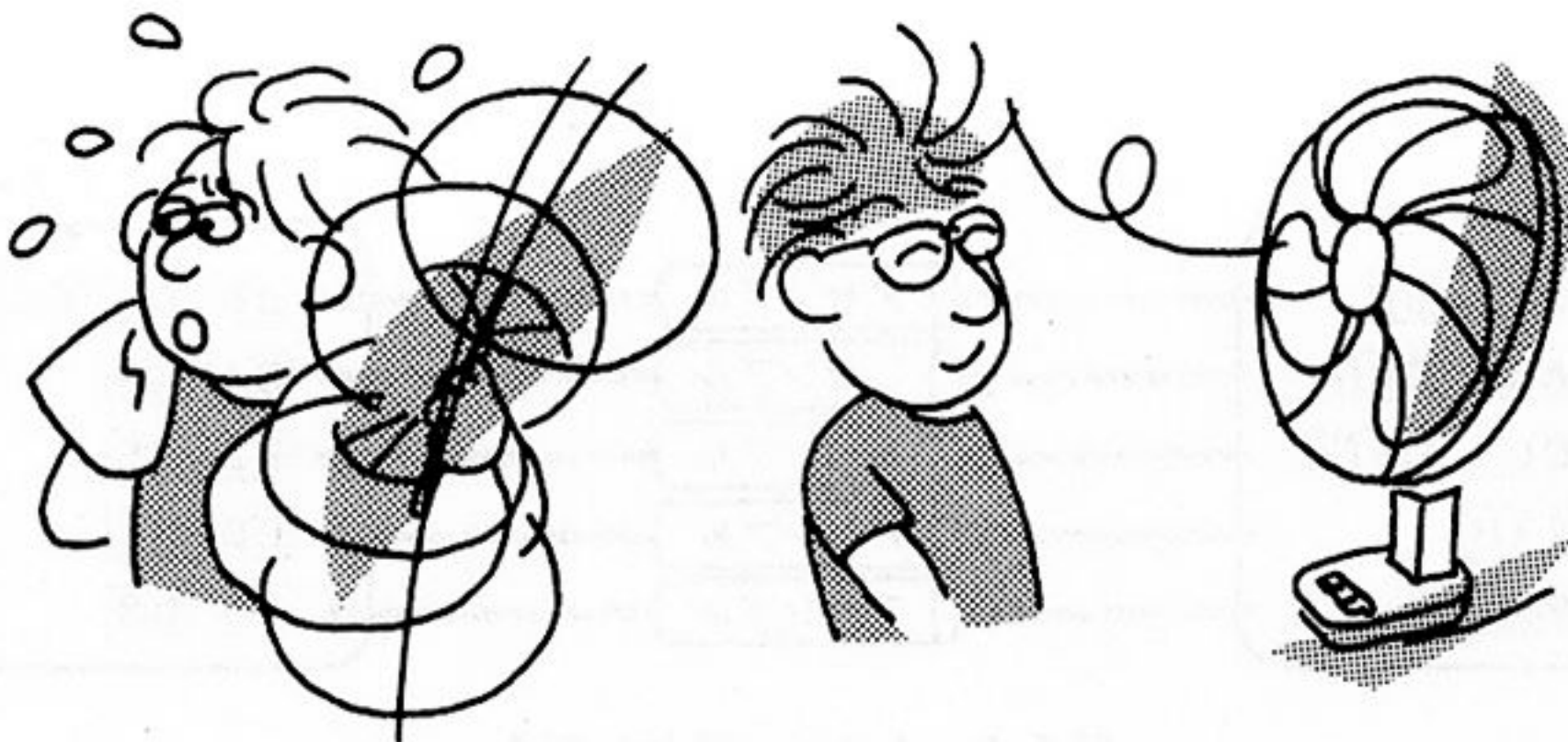
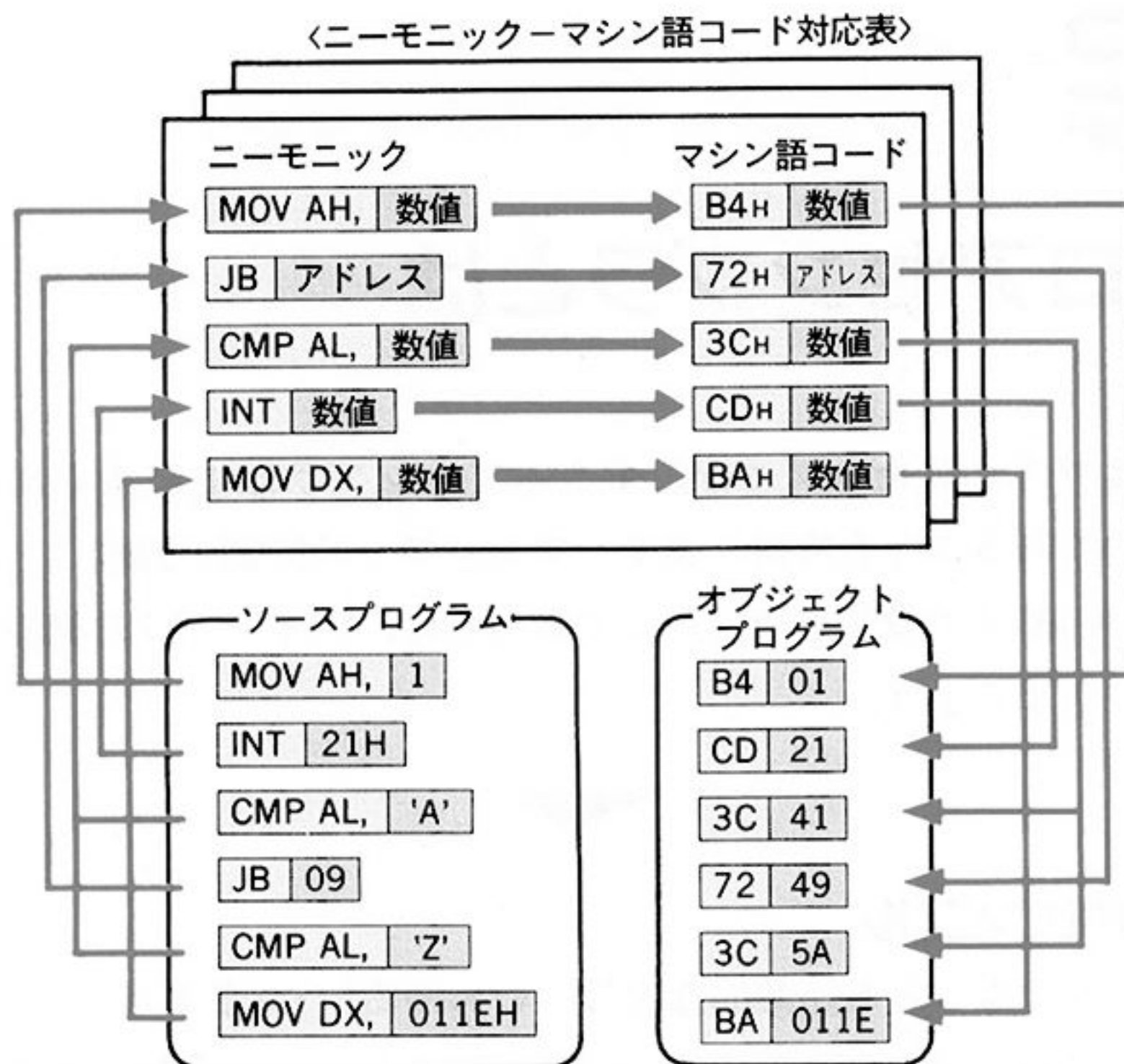



図 5-8　ハンドアセンブルの手順

1ラインアセンブラ

次に少しだけ進歩した形が「1ラインアセンブラ」と呼ばれるアセンブラです．DEBUG の A コマンドがこれにあたります．図 **5-9** のようにアセンブリ言語のニーモニックを1命令ごとにマシン語コードに自動的に変換してくれます．

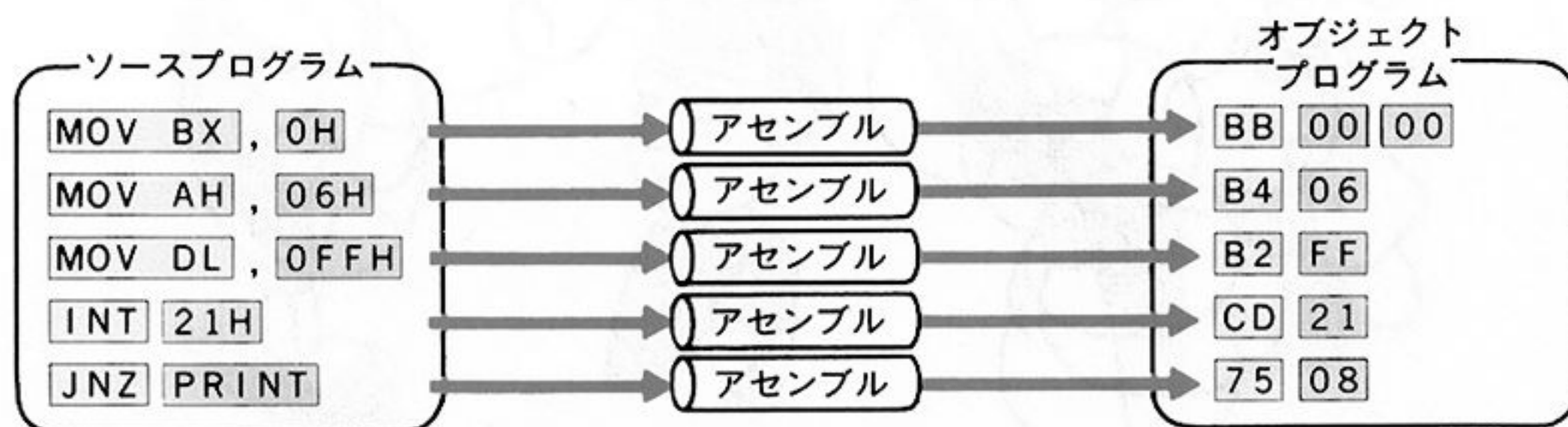



図 5-9　1ラインアセンブラ

これはハンドアセンブルにおいて，人間が対応表を引いていた作業を代わりにやってくれるプログラムということになります．ハンドアセンブルに比べればこれははるかに便利で，プログラムを作る作業が楽になります．

2パスアセンブラ

そしてその次にくるのが，これまで解説してきたような機能を持つ「2パスアセンブラ」です．1ラインアセンブラとの大きな違いは，1つにはラベルが使えること，もう1つは擬似命令が使えることです*．

ラベルを使うことによってメモリの絶対番地を扱う必要がなくなり，アドレスの概念さえわかっていればプログラムを作成することができます．また，擬似命令はメモリやセグメントの概念をわかりやすく表現したり，効率よくプログラムを組むことに役立っています．

一般にアセンブラと言えば，以上の2つの機能を持つものを指します．

マクロアセンブラ

マクロアセンブラのはっきりとした定義は一般には存在しません．本書では，次の2つの機能を持ったアセンブラを，「マクロアセンブラ」と呼ぶことにします．

- **マクロ命令**
- **分割(モジュール別)アセンブル機能**

マクロ命令とは，自分でアセンブラの命令を拡張して作ってしまう機能のことをいいます．この機能は非常に便利で，アセンブラを2倍にも3倍にも強力にすることができます．

分割アセンブルとは，ソースプログラムをいくつかの別々なファイルに分けて，それぞれを独立にアセンブルすることができる機能をいいます．でき

---

*1ラインアセンブラでもラベルを使えるものはあるが，前方参照ができないなど，完全なものではない．

あがったオブジェクトファイルをリンカでくっつけることによって実行プログラムを作成します．

マクロアセンブラにいたるまでの段階を図で示したのが図 5-10 です．次節からは，マクロアセンブラの持つ機能を具体的に解説していきます．

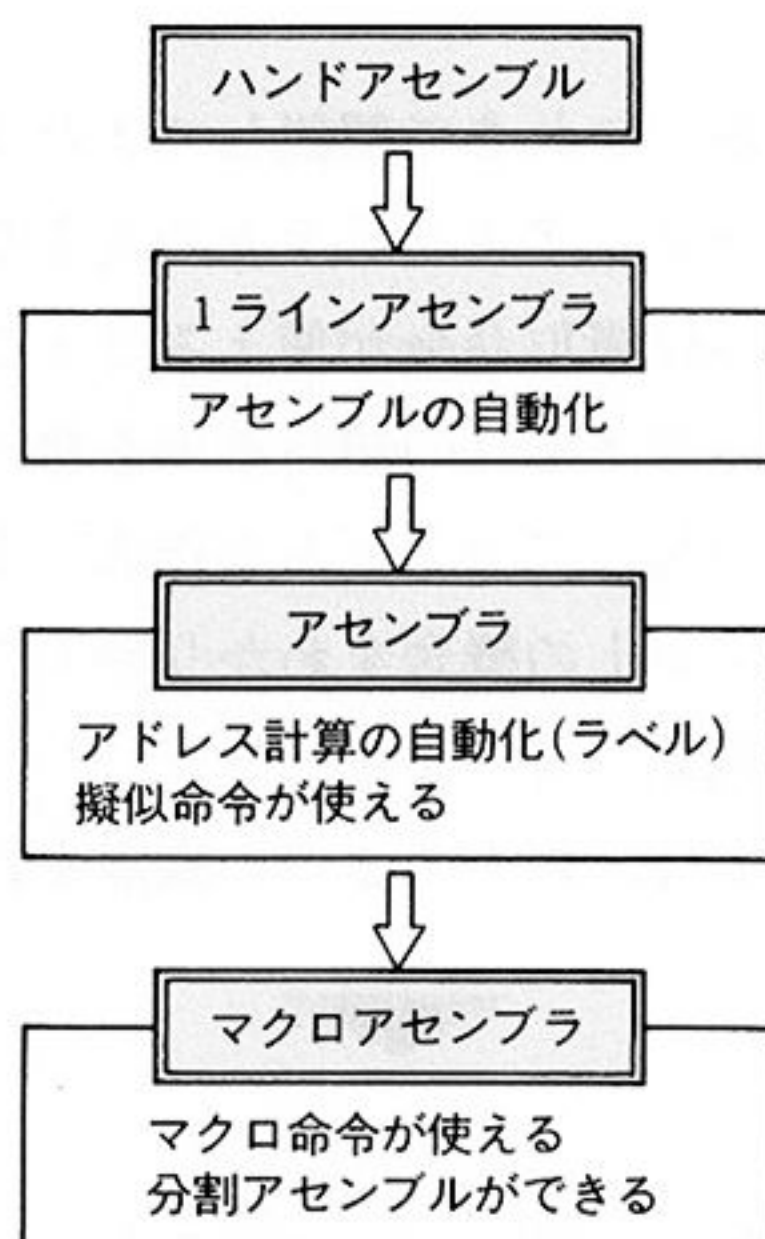



図 5-10　アセンブラの進化

5.3 マクロ命令とEQU擬似命令

マクロ命令とは

マクロ命令とは一言でいえば,「自分で命令を作ることのできる命令」です.自分で命令を作るということは,どういうことか実例を挙げて説明しましょう.

MS-DOSのファンクションコールを使って画面にメッセージを出力するには次のようにします.

```
MOV  AH, 9
MOV  DX, OFFSET  MESSAGE
INT  21H
```

メッセージを何度も出力するたびにこれだけの命令を書くのは面倒ですし,コメントでも書かないと何をやっているのかよくわかりません.メッセージを出力する「PRINT」という命令があれば,

```
PRINT  MESSAGE
```

と書くだけでメッセージを出力することができます.

MASMにはこんな命令はありませんが,マクロ命令を使うと必要な命令を「作る」ことができます.上に挙げた3行の命令にPRINTという名前を付けて,あらかじめ用意されている命令と同じように使うことが可能です.

マクロ命令とは,マシン語命令や擬似命令を組み合わせたものを新たな命令としてアセンブラに組み込んでしまうという命令です.新たにできた命令は,もはやマシン語命令でも擬似命令でもなく,あなたの作った新たな言語の命令になります.MASMのマクロ命令は単に命令の組み合わせに名前を

付けるだけでなく，もっと柔軟な定義が可能です．うまく使えばアセンブラでありながら高級言語のような記述でプログラムを書くことも可能になります．

次項では，マクロ命令のうち比較的単純なEQU擬似命令を，次節では使いこなすと非常に便利なMACRO擬似命令についてくわしく解説していきます．

EQU擬似命令(2)

［書式］ 名前　EQU　定数式
　　　　名前　EQU　ニーモニック
　　　　名前　EQU　文字列

EQU擬似命令を使って定数定義ができることは5.1節で解説しました．実はEQU擬似命令の機能はそれだけにとどまりません．EQU擬似命令によって置き換えられるものには次のものがあります．

- **定数**　……定数を名前で置き換える．
- **マシン語命令**　……マシン語命令のニーモニックを置き換える．
- **テキスト**　……テキスト(文字列)を名前で置き換える．

このうち定数については5.1節で解説したので，残りの2つについて解説しましょう．

ニーモニックの置換

マシン語命令のニーモニックを置き換えると，たとえば次のような記述が可能になります(図5-11)．

Z80CPUのアセンブラに慣れ親しんだ人ならば，この方がプログラムを書きやすいかもしれません．ニーモニックはこのように置き換えることができますから，自分の好きなニーモニックを使うことができるわけです．この機能は自分専用のアセンブラを作る機能のように思えませんか？　これがマクロ機能の1つです．

```
            INCLUDE MSDOS.H

STACKSIZ            EQU       100H
BUFSIZ              EQU       1000H

LD                  EQU       MOV
EX                  EQU       XCHG
JP                  EQU       JMP
DJNZ                EQU       LOOP
CP                  EQU       CMP

DGROUP    GROUP     DATA1,DATA2
CODE      SEGMENT
          ASSUME    CS:CODE,DS:DGROUP,ES:DATA3,SS:STACK
START:
          LD        AX,DGROUP
          LD        DS,AX
          LD        AX,DATA3
          LD        ES,AX

STORE:
          LD        BYTE PTR CHR1F,1
          LD        CHR1,AL
          JP        ROMAKANA_END
SECONDCHR:
          CALL      ISALPHA
          JZ        TRANSFER
          ;
          EX        AL,CHR1
          ;
          CALL      SETCHR
          LD        AL,CHR1
          LD        BYTE PTR CHR1F,0
          JP        SET_END
TRANSFER:
          AND       AL,5FH
          EX        AL,CHR1
          ;
          LD        SI,OFFSET DGROUP:HYOU3
          LD        CX,9
SLOOP:
          CP        [SI],AL
          JE        SFOUND
          ADD       SI,6
          DJNZ      SLOOP
          ;
```

8086CPUの命令に対して，別名として
Z80のニーモニックを定義する

図5-11 EQU擬似命令によるマシン語命令の置換

テキストの置換

テキストの置き換えとは次のようなことです.

```
COL EQU [BP−4]
```

と定義されていると，左のステートメントは右のように変換されます.

```
MOV AX, COL → MOV AX, [BP−4]
```

くわしくは6章で解説しますが，C言語などの高級言語とアセンブラのプログラムをリンクする場合にはパラメータをスタックで受け渡すことが多く，アドレッシングモードが複雑になりがちです.「BP−4」のような値を間違えることも少なくありません．そこで変数を指すアドレッシングモードの文字列に名前を付けて，その名前で参照するのです.

これは文字列を名前で置き換えてしまう機能で，さまざまな応用が考えられます．ただし，あまり複雑な定義をすると，かえってわかりにくいプログラムになってしまうので気をつけましょう*.

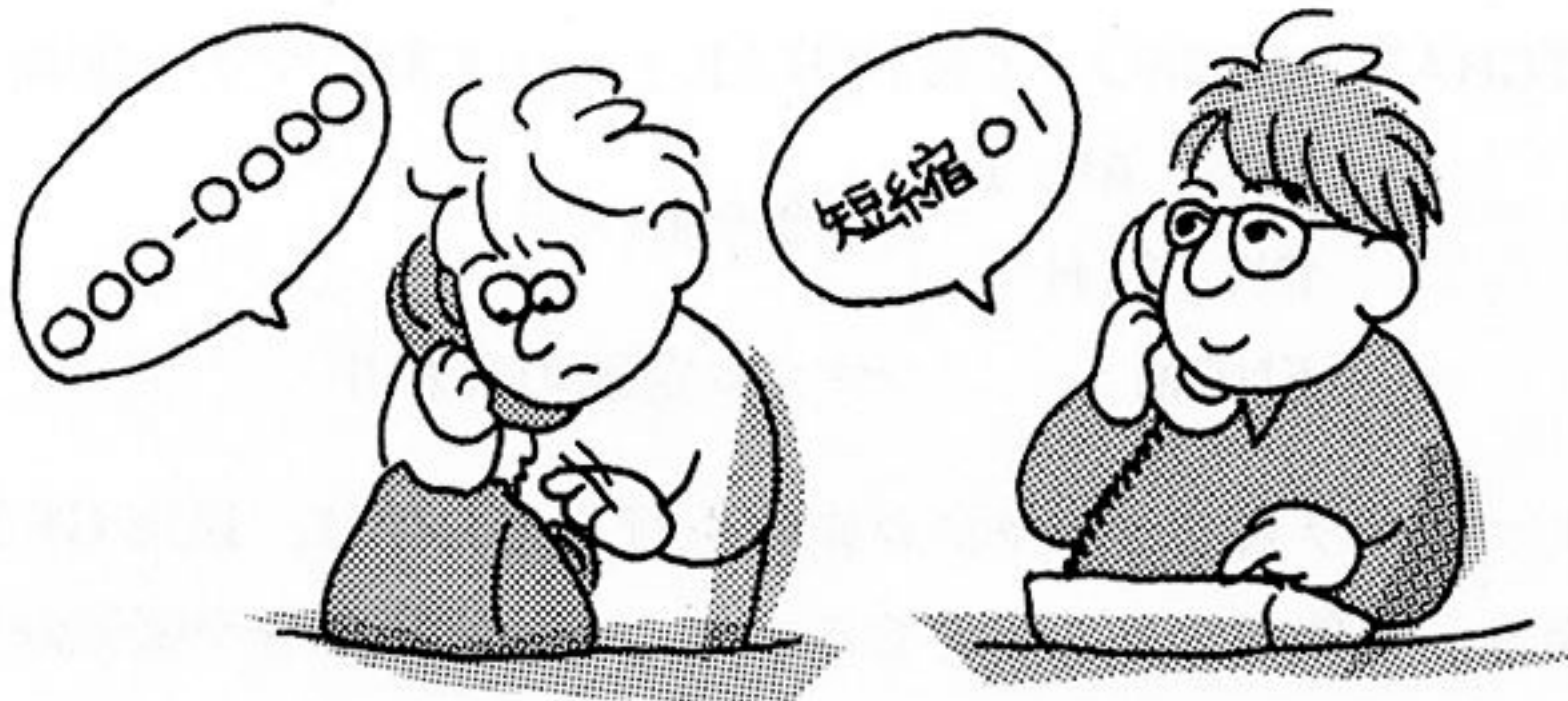
例題のプログラムでこのテキストの置き換えを使った例を図5-12に示します.

```
        INCLUDE MSDOS.H

STACKSIZ        EQU     100H
BUFSIZ          EQU     1000H

CHK_BOIN        EQU     HYOU1[BX]
GET_BOIN        EQU     HYOU2[BX]    文字列に名前を定義する
RESULT          EQU     [SI+BX+1]

DGROUP  GROUP   DATA1,DATA2
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DGROUP,ES:DATA3,SS:STACK

START:
        MOV     BX,SEG DGROUP
        MOV     DS,BX
        MOV     BX,SEG DATA3
        MOV     ES,BX
```

* Ver1.25以前のMASMではテキストの置き換えはできない.

```
;-- init buffer for read/write --
        MOV     WORD PTR IN_LEN,Ø
        MOV     WORD PTR ES:OUT_LEN,Ø
        MOV     ES:OUT_PTR,OFFSET ES:OUT_BUF

        CMP     [SI],AL
        JE      SFOUND
        ADD     SI,6
        LOOP    SLOOP
        ;
        CALL    SETCHR
        MOV     AL,CHR1
        CALL    BOIN
        JNE     ROMAKANA_END
        MOV     AL,GET_BOIN
        MOV     BYTE PTR CHR1F,Ø
        JMP     SET_END
;
SFOUND:
        XCHG    AL,CHR1
        CALL    BOIN
        JNE     CHKN
        ;
        MOV     AL,RESULT
        MOV     BYTE PTR CHR1F,Ø
        JMP     SET_END
CHKN:
        CMP     CHR1,'N'
        JNE     NOTKANA
        CMP     AL,'N'
        JNE     NOTKANA
        ;
```

プログラム中で用いたシンボルは，文字列に変換してアセンブルされる

図5-12 EQU擬似命令によるテキストの置換

5.4

MACRO 擬似命令

EQU 擬似命令は命令の一部を置き換えるだけにすぎませんが，MACRO 擬似命令は命令そのものを作ってしまう働きを持っています．しかも，柔軟な定義が可能であるため応用範囲が広く，非常に強力な機能です．ぜひ使い方を理解して活用してください．

マクロ定義とマクロ呼び出し

［書式］ マクロ名　MACRO　［仮引数, …］

　　　　　　　　　　　⋮

　　　　　　　　　ENDM

・マクロ名は｛アルファベット，@，$，＿，?，数字｝からなる文字列で，数字で始まることはできない．

MACRO 命令を使えば，命令をいくつか組み合わせて新しい命令を作ることができます．たとえばファンクションコールの1番を使ってコンソールから1文字入力する命令を GETCHAR という名前で定義してみましょう．次のようになります．

```
GETCHAR  MACRO  …GETCHARという名前のマクロ定義のはじまり
         MOV  AH, 1  ┐
         INT  21H    ┘命令を並べる
         ENDM        …マクロ定義の終わり
```

ソースプログラムでこのマクロ定義を行っておけば，以降 GETCHAR という「命令」を使用することができます．マシン語命令や擬似命令と同じく

もともと組み込まれていた命令であるかのように，GETCHAR命令を使用することが可能です．たとえば，

```
    :
GETCHAR          …マクロ命令を呼び出す
CMP  AL, 1AH
JE  EOF
    :
```

のように使用します．このように定義したマクロ命令を命令として使用することを，**マクロ呼び出し**と呼びます．

マクロ展開の仕組み

マクロ命令の原理は簡単なもので，アセンブル時にマクロ呼び出しの箇所をマクロ名に対応する定義内容で置き換えるだけです．マクロ命令を含むソースプログラムをアセンブルする手順は次のようになります．

まず，MASMはマクロ定義があると，そのマクロ名と定義の内容である命令の集まりをいったん記録します．この時点では命令の集まりを記録するだけで，アセンブルは行いません．そしてマクロ呼び出しがあると，記録していた命令の列をそこへ展開し(マクロ展開)，その命令列をアセンブルします．この様子を示したのが図5-13です．

```
Microsoft MACRO Assembler  Version 3.00                 Page    1-1
                                                                06-21-88

                        C              INCLUDE MSDOS.H
 = 000D                 C   CR                 EQU      0DH
 = 000A                 C   LF                 EQU      0AH
 = 0009                 C   FC_PUTMSG          EQU      09H
 = 003F                 C   FC_READ            EQU      3FH
 = 0040                 C   FC_WRITE           EQU      40H
 = 004C                 C   FC_END             EQU      4CH

 = 0100                     STACKSIZ           EQU      100H
 = 1000                     BUFSIZ             EQU      1000H
```

図 5-13　マクロ展開の例

マクロパラメータ

マクロ命令はパラメータをとることもできます．たとえば，ファンクションコールの２番を使ってコンソールに文字を出力するPUTCHAR命令は次のように定義できます．マクロ定義の「MACRO」の後にパラメータ「CHR」が付いていることに注目してください．

```
PUTCHAR  MACRO  CHR        …ダミー引数
         MOV  AH, 2
         MOV  DL, CHR      …ダミー引数を命令中で使用できる
         INT  21H
         ENDM
```

PUTCHAR命令を呼び出すには，マクロ呼び出しにパラメータを付けます．たとえば文字'A'をパラメータとすると，次のようになります．

```
    ⋮
PUTCHAR  'A'   …実引数
    ⋮
```

このマクロ呼び出しは，アセンブルにより次のように展開されます．

```
    ⋮
MOV  AH, 2
MOV  DL, 'A'   …ダミー引数が実引数に置き換えられる
INT  21H
    ⋮
```

マクロ定義でMACROのあとに指定するパラメータ(この場合はCHR)を**ダミー引数**(仮引数)と呼びます．ダミー引数には好きな名前を付けることができ，マクロ定義のなかで命令の一部に使用することができます．

これに対し，マクロ呼び出しで指定するパラメータ(この場合は'A')を**実引数**と呼びます．マクロ定義中で使われたダミー引数は，マクロ展開時に実引

数に置き換えられます．

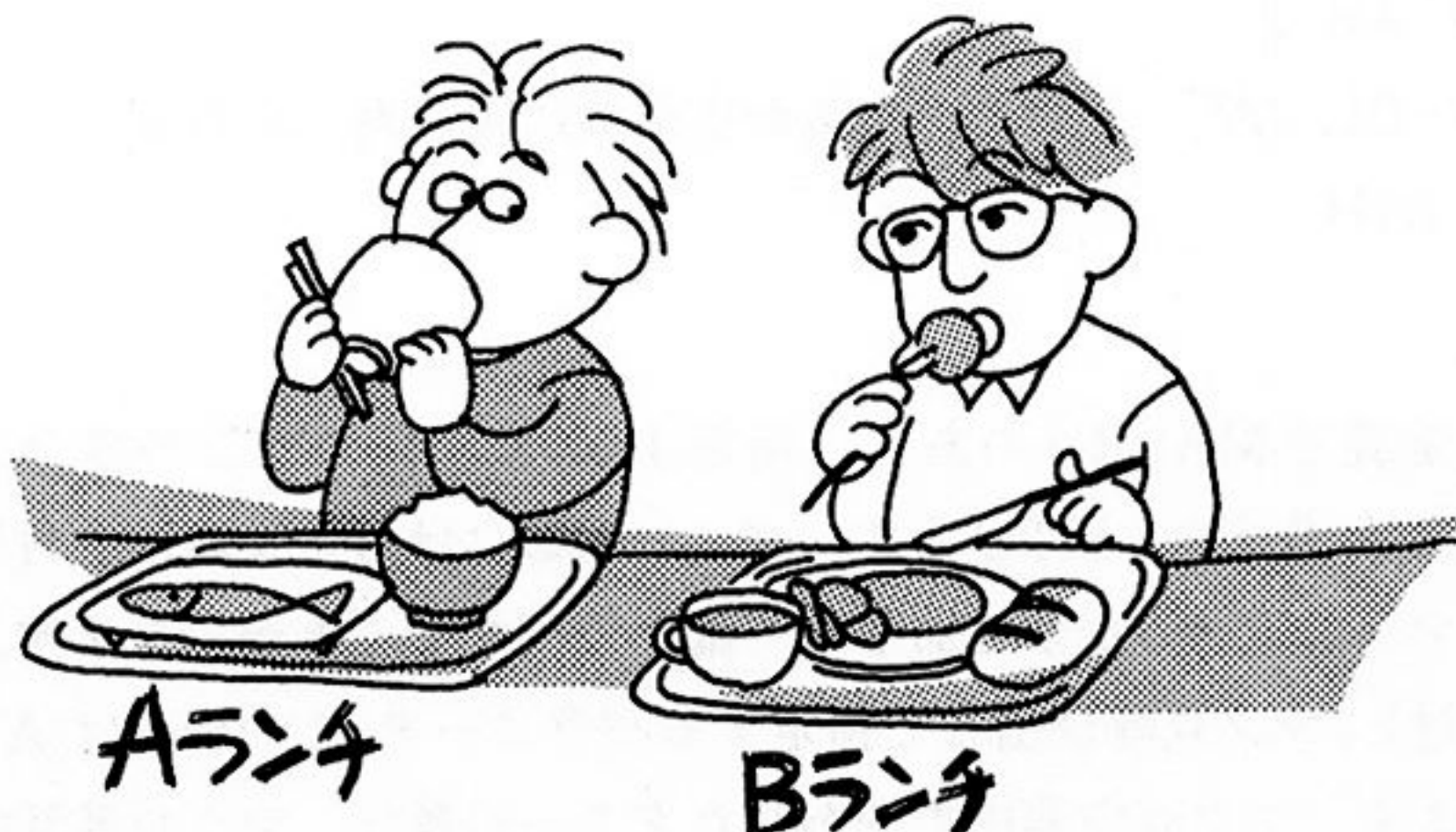
以上がパラメータ付きマクロの展開の仕組みです．

また，パラメータはいくつも付けることができます．たとえば次のマクロ定義が何をするかわかるでしょうか．

```
MOVMEM  MACRO  SOURCE, DEST, COUNT
        CLD
        MOV  SI, OFFSET  SOURCE
        MOV  DI, OFFSET  DEST
        MOV  CX, COUNT
        MOVSB
        ENDM
```

これはあるメモリ領域から他のメモリ領域へブロック転送を行うマクロ命令を定義したものです．このマクロ命令は，データラベルを使って次のように呼び出します．

```
MOVMEM  BUF1, BUF2, 200H
```

この命令によってBUF1から始まるメモリの内容がBUF2から始まるメモリへ200_Hバイト転送されます．

パラメータによる条件マクロ

さらにマクロ命令のすぐれているところは，どのようなパラメータが指定されたかによって展開される命令を選択できることです．たとえば，先のPUTCHAR命令はパラメータとしてDLレジスタが指定されると，

```
MOV  DL, DL
```

という無意味な命令を展開してしまうことになります．そこでパラメータがDLである場合には，この命令を展開しないように定義してみましょう．

```
PUTCHAR  MACRO  CHR
         MOV  AH, 2
         IFDIF  <CHR> , <DL>   ┐ 実引数とDLが等しくないと
         MOV  DL, CHR          │ きにアセンブルされる
         ENDIF                 ┘
         INT  21H
         ENDM
```

この定義には擬似命令IFDIFがありますが，これは指定した2つの文字列が等しくないときに真となる条件アセンブル擬似命令です．この場合，実引数の文字列とDLという文字列が等しくないときだけ，ENDIFまでのブロック内がアセンブルされます(ダミー引数CHRとDLを比較するのではないことに注意)．逆に実引数の文字列とDLが等しい場合には，ブロック内はアセンブルされません．

この定義により，「PUTCHAR DL」は，

```
MOV  AH, 2
IFDIF  <DL> , <DL>
MOV  DL, DL
ENDIF
INT  21H
```

と展開され，〈DL〉と〈DL〉は等しいので条件擬似命令 IFDIF は偽となり，不用な命令「MOV DL,DL」はアセンブルされません．

マクロ定義のなかで条件アセンブル擬似命令を使うことによって，このようにパラメータの種類に柔軟に対応できる命令を作ることができます．IFDIF 擬似命令のほかにも，文字列を扱う条件擬似命令が用意されており，マクロ命令の定義に利用することができます（194 ページの表5-1 を参照）．

LOCAL 擬似命令

マクロ定義を利用する際に，1つ注意しておかなければならないことがあります．それはマクロ定義におけるラベル等の名前の定義に関することです．たとえば，指定された回数だけコンソールに空白を出力するマクロ SPACE は，次のように定義することができます．

```
SPACE   MACRO   COUNT
        MOV  CX, COUNT
PUTSPACE:
        MOV  AH, 2
        MOV  DL,' '
        INT  21H
        LOOP  PUTSPACE
        ENDM
```

この例では，マクロ定義のなかで PUTSPACE というラベルが定義されています．マクロ定義の時点では，このラベルはマクロ定義の内容として記録されるだけで，アセンブルはされないので，ラベルとして定義されたことにはなりません．マクロ呼び出しによって展開されたときに，はじめてラベルとして定義されたことになります．

このマクロ命令をもう一度呼び出すと，PUTSPACE というラベルも，もう一度展開されて定義されることになります．同じ名前のラベルは2回以上定義することはできませんから，これはエラーになります．

そこでマクロ定義のなかでラベルを定義するには，マクロ展開のなかだけで有効になるようにLOCAL擬似命令を使用します．LOCAL擬似命令はマクロ定義のはじめで使用し，マクロ定義のなかであらたに定義する名前を指定します．

```
SPACE   MACRO   COUNT
        LOCAL   PUTSPACE   …マクロ定義のなかだけで有効なラベル
        MOV  CX, COUNT
PUTSPACE:
        MOV  AH, 2
        MOV  DL, ' '
        INT  21H
        LOOP  PUTSPACE
        ENDM
```

このように定義することによって，PUTSPACEは何度展開されても展開されたブロック内だけで有効になり，2重定義になることがありません．

マクロ内では，ラベル以外にもEQU擬似命令によって定義する定数なども2重定義とならないようLOCAL擬似命令が必要です．

マクロ命令とプロシージャの違い

マクロ命令の使い方は，ほぼ理解できたと思います．ここで，マクロ命令の特徴をもっとよく理解するために，マクロ命令とプロシージャの違いを解説します．

マクロ命令とプロシージャ(サブルーチン)は一見よく似ています．あるところで本体を定義しておいて，別な場所でそれを呼び出すことができる，という点では同じです．しかし，以下に解説するようにサブルーチンとマクロは大きな違いがあるのです．

展開とコール

プロシージャは実体が1つしかありません.呼び出されるときには,CALL 命令を使って呼び出します．どこで何回呼ばれようとも，たった1つのプロシージャが繰り返し呼び出されるにすぎません．ですから何度呼び出してもCALL 命令の分しかメモリを消費しません.

これに対してマクロ命令は，呼び出されるたびに定義された内容が展開されます．マクロ命令が何度も呼び出されると，同じ命令の並びが何度も繰り返されることになるのです．したがってマクロ命令は，プロシージャよりも多くのメモリを消費してしまうことになります.

ただし，プロシージャはCALL 命令で呼び出されるので，そのたびにスタックにアドレスを積んでRET 命令でそれを復帰するという過程が必要なのに対し，マクロ命令では続けて展開された命令が実行されるので，実行はマクロ命令の方が高速です.

図 5-14 は両者の違いを図解したものです.

パラメータの展開

パラメータを持つマクロ命令は，呼び出されるときのパラメータによって異なる命令が展開されることになります．たとえば，先のPUTCHAR ならば,「PUTCHAR 'A'」と「PUTCHAR BL」では，マクロ定義中の「MOV DL,CHR」という命令が，それぞれ，

```
MOV  DL, 'A'
MOV  DL, BL
```

というまったく異なった命令に置き換えられてしまいます.

さらに前述したように，条件擬似命令を利用すればパラメータの種類によって展開される命令列を選択することができます．この場合，展開後のプログラムはまったく違うものになることもありえます.

このように，パラメータによって展開されるプログラムの実体が異なるという点が，マクロ命令とプロシージャの大きな違いです.

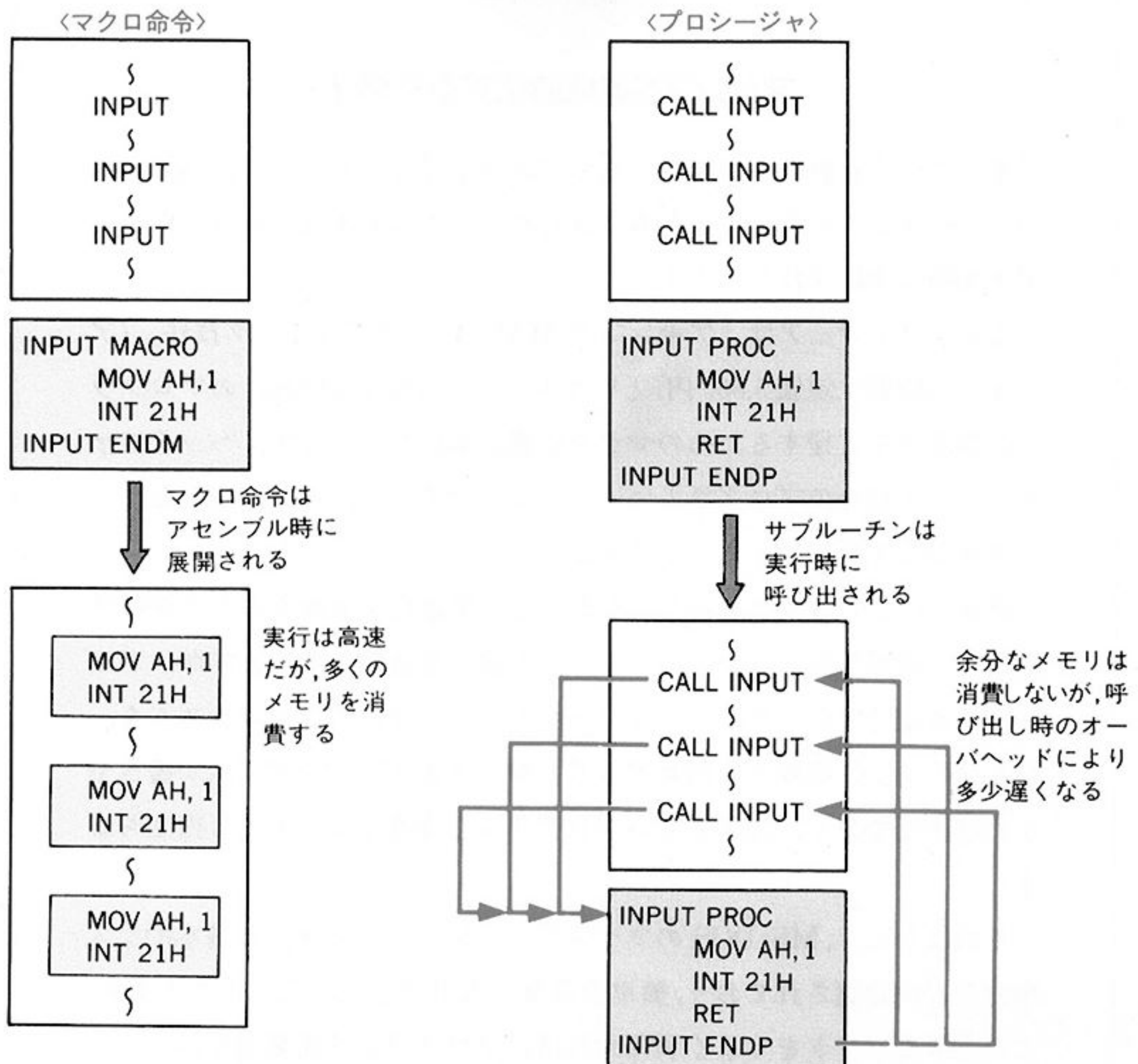



図 5-14 マクロ命令とプロシージャの違い

COLUMN

アセンブラを構造化するマクロ

本文でマクロ命令を活用して自分で命令を作ることにより，高級言語的な記述さえも可能になると書きましたが，これを実現したソフトウェアが実際に市販されています．

このソフトウェアは「アセンブラMACROプログラミング技法」（アスキー出版局：定価5,800円）というもので，内容はMASMのプログラムの構造化を支援するためのマクロ定義を集めたものです．ヘッダファイルとして自分のプログラムにインクルードすれば，そこに定義されたマクロ命令を利用することができます．

提供されているマクロ命令のなかでも，構造化を支援するものが非常に強力で利用価値の高いものです．右の図に記述例を示しますが，アセンブリ言語ではわかりにくくなりがちなプログラムのアルゴリズムを，C言語のように論理的に記述することができます．プログラムが書きやすいばかりでなく，読みやすいプログラムを作成することにも役立ちます．

そのほかにも，MS-DOSのファンクションコールを使った基本的な入出力などが定義されており，簡単な命令で入出力を行うことができます．このようなソフトをうまく利用すれば，プログラムの開発効率はぐんと高くなることでしょう．

アセンブラMACROプログラミング

```
        include amscls.inc  ……………構造化マクロを定義したヘッダファイル
                                    をインクルードする
        $_init  GEN

svctxt          macro
                irp     register,<si,ax,bx,cx,dx,di,ds,es>
                push    register
                endm
                endm

        $_while_ <cmp   byte ptr es:[di],0Ah>,NE
                mov     cl,es:[di]
                putchar cl
                $_switch_ …………C言語と同じ制御構造が実現できる
                $_case_ <cmp cl,0Dh>,E
                        $_break_
                $_case_ <cmp cl,'#'>,E
                        mov     pflag,1
                        inc     di
                        mov     cl,es:[di]
                        putchar cl…………C言語の標準関数に似た命令が使える
                        $_switch_
                        $_case_ <cmp cl,'0'>,E
                                $_if_ <cmp v0siz,0>,E
                                        mov v0siz,GSIZE*2
                                $_else_
                                        puts    errmsg0
                                        pop     es
                                        pop     bx
                                        jmp     generr
                                $_endif_
                                $_break_
                        $_case_ <cmp cl,'1'>,E
                                $_if_   <cmp v1siz,0>,E
                                        mov v1siz,GSIZE*2
                                $_else_
                                        mov     errno,'1'
                                        puts    errmsg0
                                        pop     es
                                        pop     bx
                                        jmp     generr
                                $_endif_
                                $_break_
                        $_default_
                                puts    errmsg1
                                pop     es
                                pop     bx
                                jmp     generr
                        $_endswitch_
                        $_break_
```

分割アセンブルの概念

5.2節で解説したように，MASMでは分割アセンブルを行うことが可能です．分割アセンブルはプログラミングの効率を大きく向上させる，モジュール別プログラミングへとつながります．プログラムのモジュール構造は高級言語の世界では常識ともいえる概念ですが，MASMにおいてもやはり重要な概念の1つです．

本節では分割アセンブルとその効果について解説し，そのあとモジュール別プログラミングをサポートするMASMの機能について解説します．

分割アセンブルとは

分割アセンブルとは，ソースプログラムを複数のソースファイルに分けて，それぞれを独立にアセンブルすることです．アセンブルの結果出力されるオブジェクトファイルも複数になりますが，それらをつなぎ合わせることにより1つの実行ファイルができあがります．一見手間のかかる方法のように思えますが，分割アセンブルが可能であることによるメリットはその手間をはるかに上回るものです．

一般には，プロシージャ部分を取り出して独立なソースファイルにすることによって，ソースプログラムを分割します．すると，1つはプロシージャの呼び出しはあってもプロシージャの本体がないソースファイル(**メインモジュール**)，もう1つはプロシージャの本体はあってもプロシージャの呼び出しがないソースファイルになります．

それぞれのソースファイルは，それ自身アセンブル可能でなければなりません．すなわち，セグメントの定義(SEGMENT～ENDS)やセグメントレジスタの割り当て(ASSUME)など，アセンブルに必要な指示を擬似命令によって正しく与えてやる必要があります．ただし，メインモジュール以外の

ソースファイルでは，END擬似命令で実行開始アドレスを指定しないことに注意してください(71ページ参照)．

さて，1つのソースファイル(.ASM)をアセンブルすると，1つのオブジェクトファイル(.OBJ)が生成されます．そして4章で触れたように，それをつなぎ合わせて1つの実行ファイルを作成するのがリンクの操作になります．リンクの操作の役割は，異なるオブジェクトファイルに分割されている，プロシージャの本体とプロシージャの呼び出しをうまくつなげることにあります．

以上の分割アセンブルの仕組みをまとめたのが図5-15です．

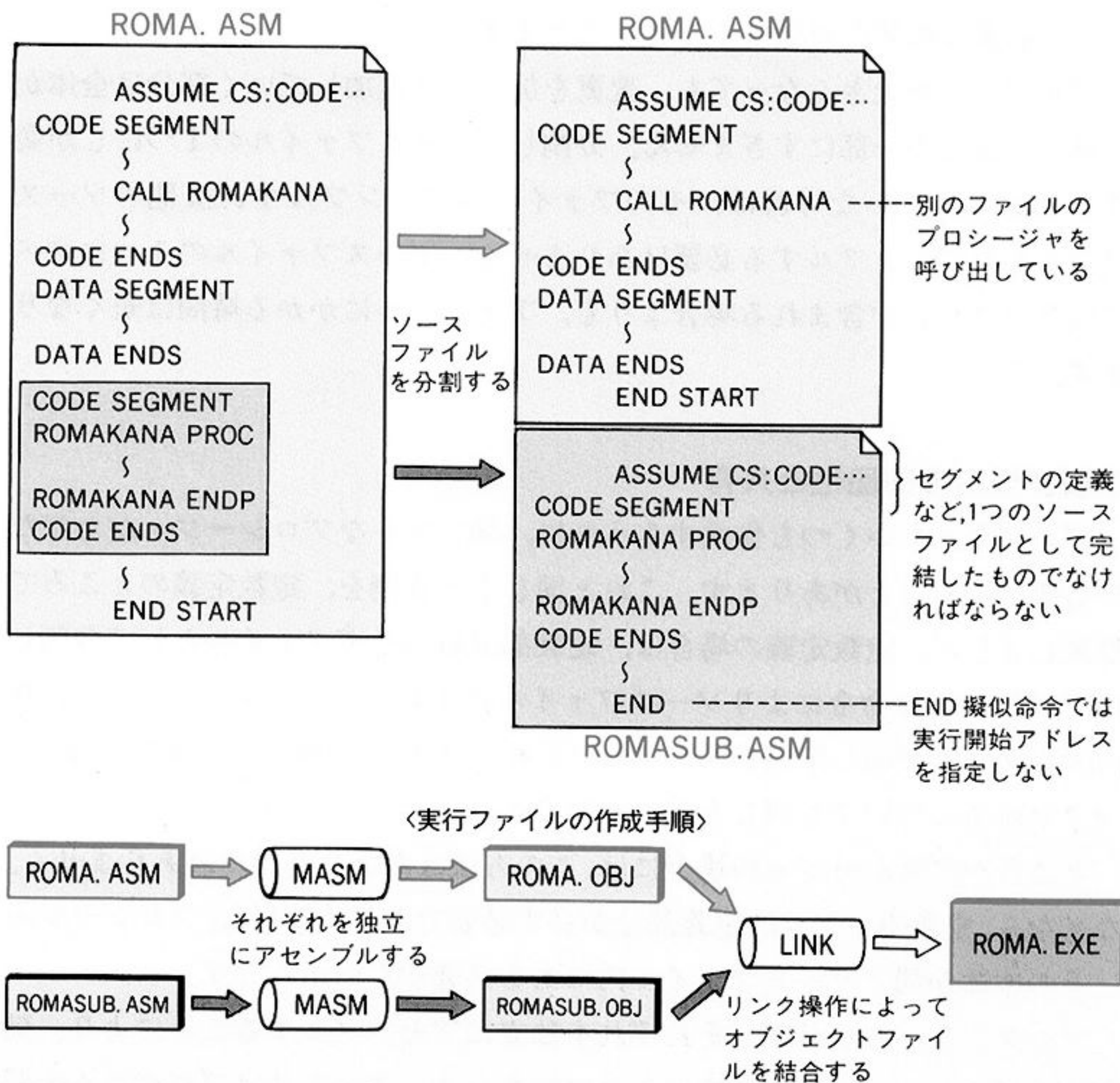



図5-15 分割アセンブルの仕組み

分割アセンブルの利点

分割アセンブルの利点を以下にいくつか示していきます.

アセンブル時間の短縮

分割アセンブルの第一のメリットは，プログラムの開発時間を短縮できることです．プログラムを作成するためには，アセンブル&リンクを何度も行わなければなりません．なぜなら，プログラムを実行してみて思いどおりに動かないところを修正したり，気に入らないところを改良したりすることを何度も繰り返すからです．ソースファイルを分割することにより，このサイクルに必要な時間を短縮することができます．

プログラムが大きくなっても，変更を加えたり追加していく部分は全体から見ればほんの一部にすぎません．分割したソースファイルの１つにしか変更が加えられていなければ，そのファイルをアセンブルすれば他のソースファイルをアセンブルする必要はありません．ソースファイルの１つにすべてのプログラムが含まれる場合よりも，アセンブルにかかる時間は短くなります．

プログラムの部品化と共有

プログラムをいくつも作成するうちに，同じようなプロシージャがたびたび必要になることがあります．これと同じような例を，定数定義のところで解説しました．定数定義の場合は，定義部分をヘッダファイルとして分割しINCLUDE擬似命令によりソースファイルにインクルードすることで，複数のプログラムで同じ定義を共有することができます（180ページ参照）．また，マクロ命令についても同じ方法が有効です．

ところがプロシージャの場合には，この方法はあまり有効ではありません．なぜなら，定数やマクロは定義部分が必ず必要であるのに対し，プロシージャはその本体が同じソースファイル内にある必要がないからです．

ソースファイルを分割しそれぞれを独立にアセンブルすることにより，効率よくプロシージャを共有することができます．この方法は**プログラムの部**

品化といってもよいでしょう．プロシージャという部品を利用しながらプログラムを組み立てていくという考え方です．

この仕組みを図5-16に示します．分割アセンブルができなければいろいろなプログラムに同じような部分が重複して存在し，プログラム作成の労力もアセンブルの時間も無駄になってしまいます．分割アセンブルが可能であれば，部品は一度アセンブルするだけで，あとはオブジェクトファイルをリンクすれば利用することができます．

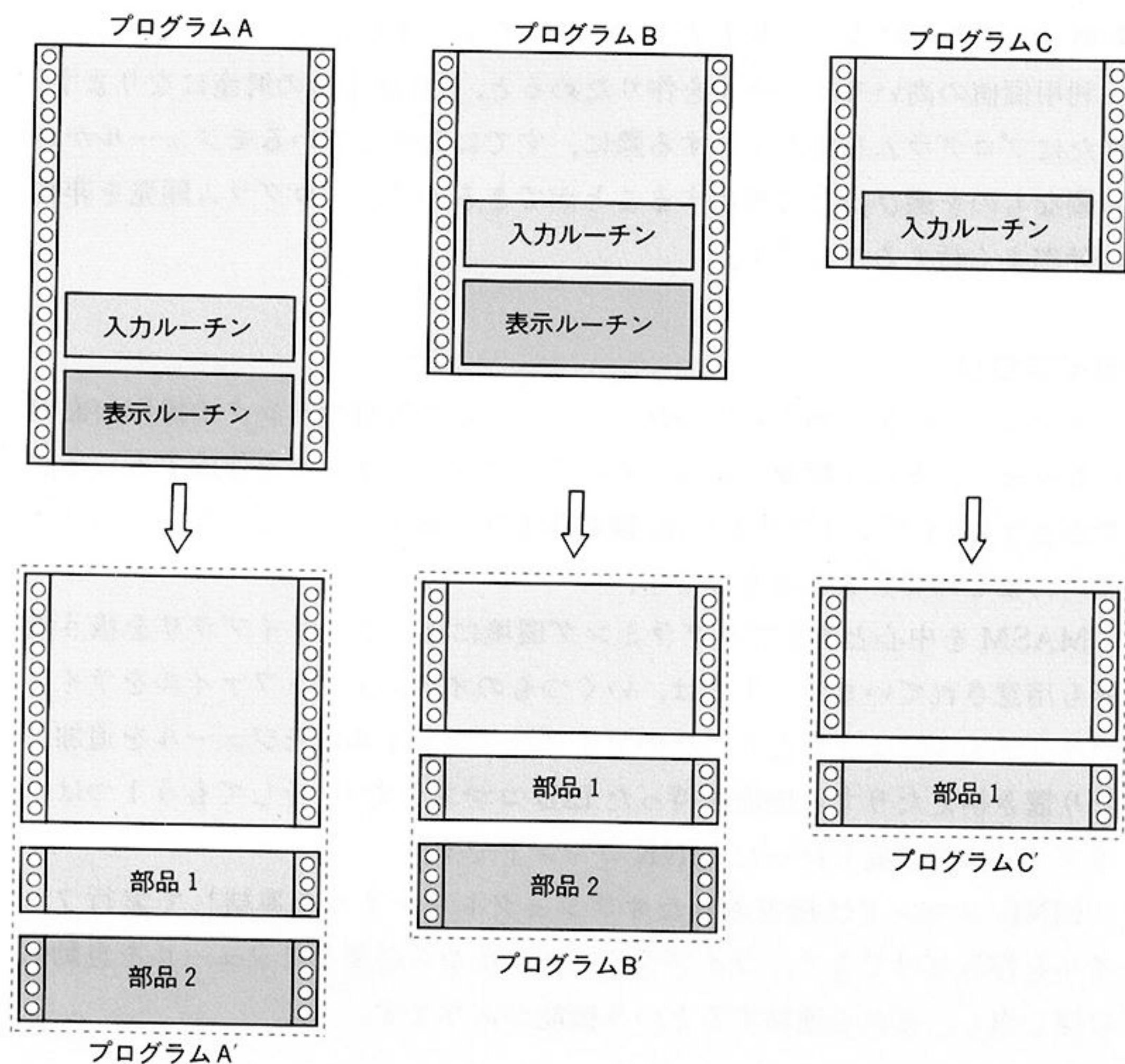



図5-16 プログラムの部品化

モジュール

プログラムの部品化と共有の考えをさらに押し進めたものがモジュール化の考え方です．モジュール化は，部品の独立性を高めることによってプログラムの開発効率と信頼性を高めようとする考え方です．

1つのモジュールは密接に関連する機能を持つプロシージャの集まりからなります．しかも，全体としては他のモジュールに依存したり影響を与えることが少なくなければなりません．このようなモジュールを部品として利用すれば，信頼性の高いプログラムを効率よく作成することができます．それぞれの機能がはっきりしているため部品として利用しやすく，部品どうしが影響しあうことによって発生するバグが少なくてすむからです．

利用価値の高いモジュールを作りためると，それは1つの財産になります．新たにプログラムを作ろうとする際に，すでに完成しているモジュールから必要なものを選び出して利用することができるので，プログラム開発を非常に効率よく行えるからです．

ライブラリ

モジュールの数が増えるにつれて，ファイルの管理やリンクの操作が面倒になります．それを簡単にするためにライブラリファイルを作成することができます．ライブラリファイルは，複数のオブジェクトファイルを1つのファイルにまとめてしまったものです．

MASMを中心とするプログラミング環境には，このライブラリを扱う機能も用意されています．1つは，いくつものオブジェクトファイルをライブラリファイルにまとめたり，そのライブラリファイルにモジュールを追加したり置き換えたりする機能を持ったLIBコマンドです．そしてもう1つはライブラリ検索機能を持ったLINKコマンドです．

LINKコマンドは指定されたオブジェクトファイルを連結して実行ファイルを作るだけでなく，ライブラリファイルから必要なモジュールを自動的に探し出し，それを連結するという機能があります．

高級言語とのリンク

他のモジュールのプロシージャを呼び出す仕組みは，高級言語でもほぼ同じです．したがってその仕組みをそのまま利用して，高級言語からアセンブリ言語で作成したプロシージャを呼び出すことができます．この方法はC言語などの高級言語とマシン語プログラムをリンクする，ごく一般的な方法として活用されています．

今後アセンブリ言語だけでプログラムを作成することはほとんどなくなり，どうしても必要な部分だけをアセンブラで作成して高級言語とリンクするという方法が最も有効になるでしょう．分割アセンブルは，この方法を実現するためになくてはならない機能といってもよいでしょう．

なお本書の6章では，C言語からマシン語のプロシージャを呼び出す例題プログラムを紹介しています．

5.6 PUBLIC擬似命令とEXTRN擬似命令

分割アセンブルの概要を理解できたところで，以下の節では実際のプログラミングでこれを実現するために必要な知識を解説しましょう．まず，分割アセンブルを行うために知っておかなければならない擬似命令PUBLICとEXTRNを紹介します．

PUBLIC擬似命令

［書式］ PUBLIC　ラベル名
　　　　PUBLIC　プロシージャ名

モジュール内部で定義したラベルやプロシージャ名は，そのモジュール内だけで有効です．したがって他のモジュールで定義されたプロシージャを参照することはできません．また2つのモジュールで同じ名前のラベルを定義しても，2重定義にはならず，お互いにまったく影響しません．このことを，ラベルが**ローカル**(LOCAL：局所的)であるといいます．

しかしこのままでは，他のモジュールで定義されたプロシージャを呼び出すことができません．それを可能にするためにはプロシージャ名を定義されたモジュール以外でも有効にする，つまり**パブリック**(PUBLIC：大域的)にしなければなりません．ラベル名やプロシージャ名をパブリックにするための擬似命令がPUBLIC擬似命令です．PUBLIC擬似命令は一種の宣言であり，指定したラベル名やプロシージャ名が他のモジュールからも呼び出されることをMASMに指示することになります．

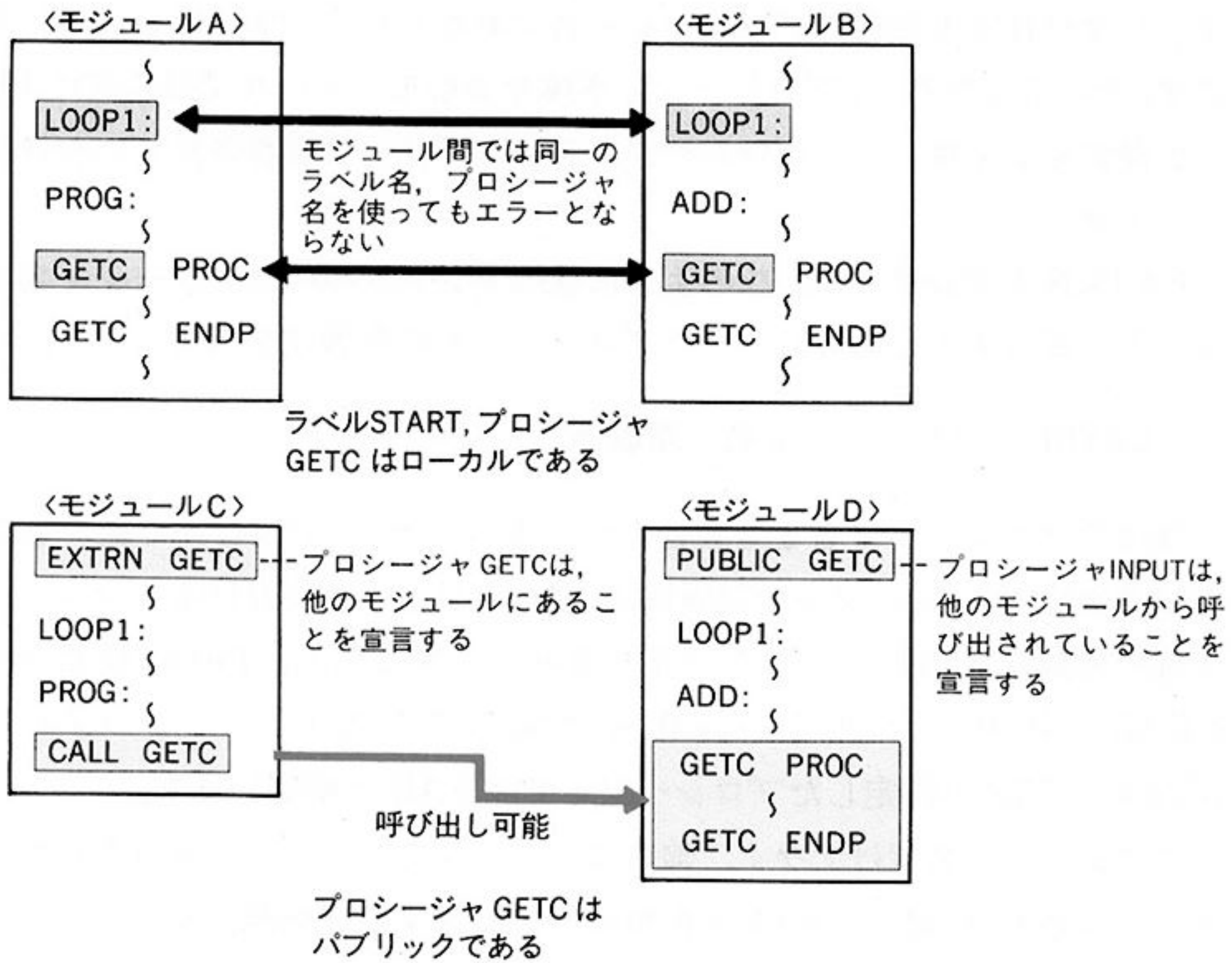



図 5-17 LOCAL なプロシージャと PUBLIC なプロシージャ

EXTRN 擬似命令

［書式］ EXTRN ラベル名
EXTRN プロシージャ名

他のモジュールのプロシージャを呼び出す方法は，同じモジュール内にあるプロシージャを呼び出す方法とまったく同じです．すなわち，

```
CALL プロシージャ名
```

のようにプロシージャ名をそのまま使って呼び出すことができます．

ただし，EXTRN(EXTeRNal：外部の)擬似命令で他のモジュールにあるプロシージャ名であることをMASMに指示しておかなければなりません．

そうしなければ未定義のプロシージャ名であるとしてエラーになってしまいます．そしてもちろん，プロシージャ本体を含むモジュールでは前述のPUBLIC擬似命令を使って，プロシージャ名がパブリック宣言されていなければなりません．

EXTRN擬似命令では，次のように他のモジュールのプロシージャ名であることを宣言すると同時に，そのプロシージャの型属性を指示します．

EXTRN　プロシージャ名：型属性

複数のプロシージャ名と型属性のペアをカンマ(,)で区切って並べることもできます．プロシージャの型属性は，他のモジュールにおけるプロシージャ定義の型属性と同じでなければなりません．すなわち，PROC擬似命令でNEARと指定した(あるいは何も指定していない)プロシージャではNEAR，FARと指定したプロシージャではFARと指定します．

プロシージャ名だけでなく，他のモジュールでパブリック宣言されたデータラベル名も，同様にEXTRN擬似命令で宣言すれば参照することができます*．

この場合もやはり型属性を正しく指示しなければなりません．データラベルの型属性はデータラベルに対応するデータ定義擬似命令の型と一致させます(表5-2を参照)．

宣言されたプロシージャ名/ラベル名の型	指定する型属性
コードラベル名，またはプロシージャ名でNEAR型	NEAR
プロシージャ名でFAR型	FAR
データラベル名でBYTE型	BYTE
データラベル名でWORD型	WORD

表5-2　プロシージャ名／ラベル名と型属性

*もちろんコードラベルを参照することも可能であるが，他のモジュールのルーチンはプロシージャとして呼び出すのが一般的である．

PUBLIC 擬似命令はどこで使用してもかまいませんが，EXTRN 擬似命令は宣言する位置に注意しなければなりません．EXTRN 擬似命令をセグメント内部で使用すると，指定したプロシージャ等はそのセグメントと同じセグメント内に定義されていると解釈されます．したがってプロシージャやデータラベルの属するセグメントが異なるセグメントである場合や，不明である場合には，セグメントの外で宣言します．このことを示したのが次の図 5-18 です．

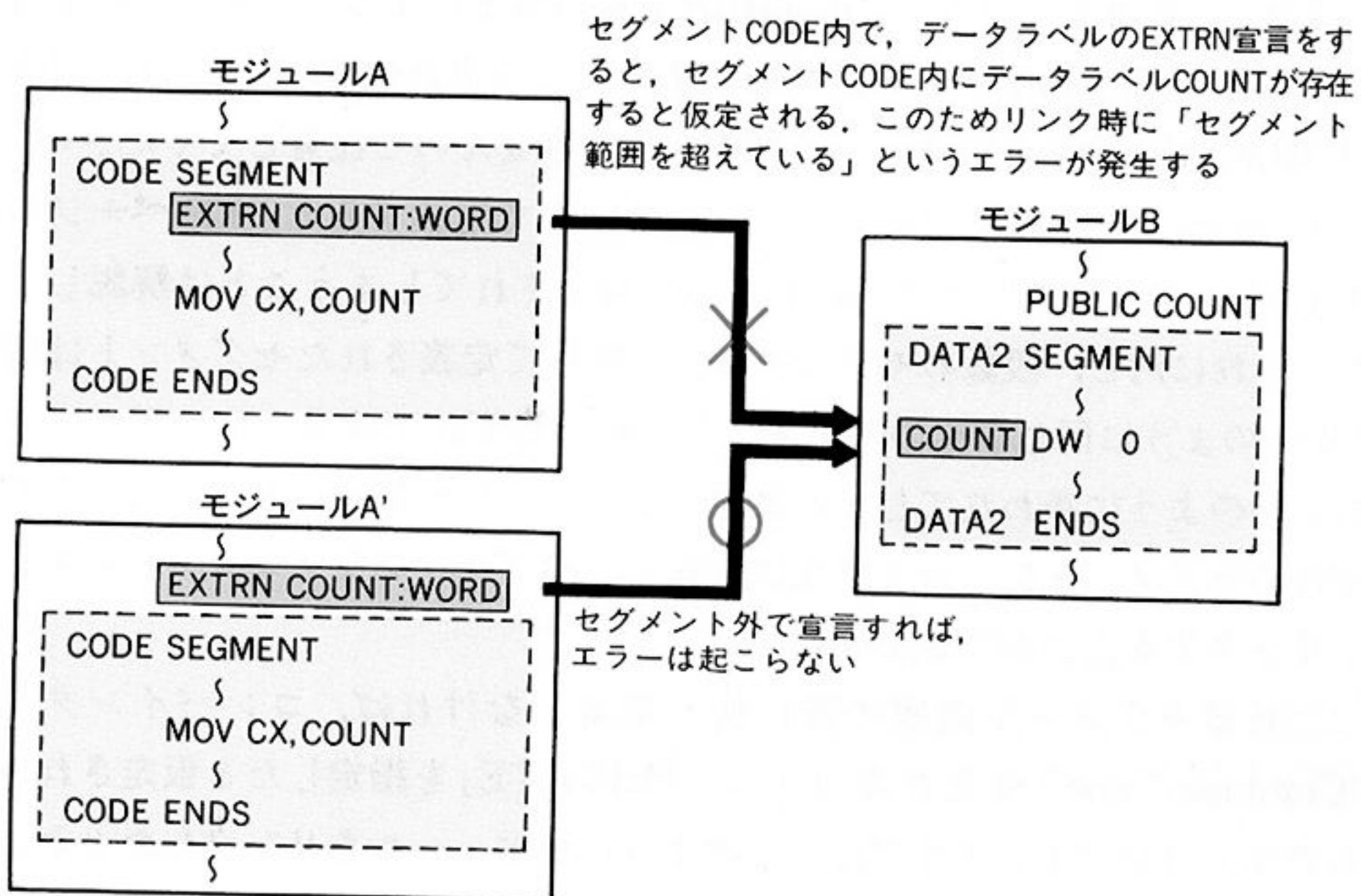



図 5-18　EXTRN 擬似命令の宣言位置の問題

セグメントの結合(コンバインタイプ)

［書式］　セグメント名　SEGMENT　コンバインタイプ
　　　　　　　　　　　　　⋮
　　　　　セグメント名　ENDS

分割アセンブルにより2つ以上のモジュールをリンクしてプログラムを作成する場合には，セグメントがどのように結合されるかに注意しなければなりません．結論からいうと，SEGMENT擬似命令によるセグメントの定義でPUBLIC属性を指定する必要があります．この場合のPUBLICは，前節のPUBLIC擬似命令とはまったく関係がありませんので注意してください．

1つのモジュール内でセグメントを分割して定義すると，129ページの図4-17のように1つのセグメントとして結合されてしまうことは解説しました．これに対し，複数のモジュールに分割して定義されたセグメントは図5-19-aのように同じ名前のセグメントでもそれぞれが独立したセグメントであるかのように扱われてしまいます．したがってオフセットアドレスもそれぞれのセグメントについて独立に存在し，パブリックなプロシージャを正しくリンクすることができません．

これはセグメント定義の際に何も指定しなければ，コンバインタイプ(Combine Type：結合方式)として「PRIVATE」を指定したと仮定されるからです．コンバインタイプは，このようにモジュールをリンクしたときにセグメントをどのように結合するかを指定する属性です．

同じ名前のセグメントが正しく結合されるためには，コンバインタイプとして「PUBLIC」を指定しなければなりません．そうすれば図5-19-bのように同じ名前のセグメントは1つに結合されます．

なお，スタックセグメントを定義するためにSEGMENT擬似命令で指定した「STACK」も実はコンバインタイプの1つです．STACKを指定すると，PUBLICと同じようにセグメントは1つにまとめられ，しかもスタックセグメントとして扱われます．スタックセグメントの情報は4章で解説したようにEXEヘッダに伝えられるので，コンバインタイプの指定が必要なのです．

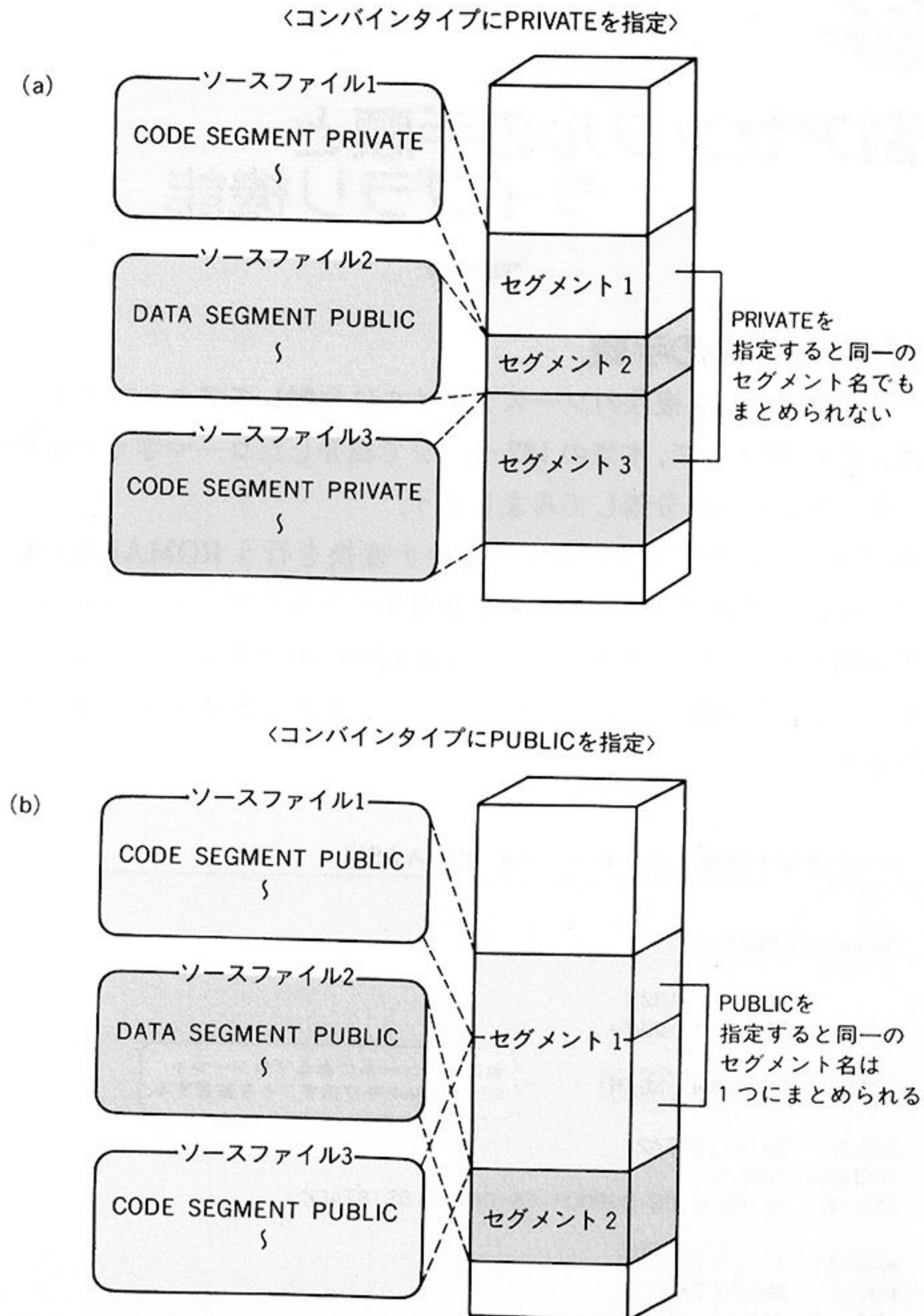



図5-19　コンバインタイプ

5.7

分割アセンブルの手順とライブラリ機能

分割アセンブルの手順

ソースプログラムを複数のソースファイルに分割してアセンブルする手順を解説します．例として，本章の 182 ページで紹介したローマ字カナ変換プログラムをモジュールに分割してみましょう．

このプログラムのなかで，ローマ字カナ変換を行う ROMAKANA プロシージャは他のプログラムでもそのまま利用できるので，１つのモジュールとして分割することにします*．そしてそれ以外の部分をメインモジュールとします．こうして分割した 2 つのモジュールを**リスト 5-2** および**リスト 5-3** に示します．

リスト 5-2　ローマ字カナ変換メインモジュール ROMA.ASM

```
        INCLUDE MSDOS.H

STACKSIZ        EQU     100H
BUFSIZ          EQU     1000H

        EXTRN   ROMAKANA:NEAR

DGROUP  GROUP   DATA1,DATA2
CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE,DS:DGROUP,ES:DATA3,SS:STACK

EXIT    MACRO
        MOV     AH,FC_END
        INT     21H
        ENDM
```


＊6 章ではこのモジュールを利用して，デバイスドライバを作成している．

```
READ_STDIN      MACRO
        MOV     BX,Ø
        MOV     AH,FC_READ
        INT     21H
        ENDM

WRITE_STDOUT    MACRO
        MOV     BX,1
        MOV     AH,FC_WRITE
        INT     21H
        ENDM

START:
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     AX,DATA3
        MOV     ES,AX
;-- init buffer for read/write --
        MOV     WORD PTR IN_LEN,Ø
        MOV     WORD PTR ES:OUT_LEN,Ø
        MOV     ES:OUT_PTR,OFFSET ES:OUT_BUF
M_LOOP:
        CALL    GETC
        JC      FLUSH
        CALL    ROMAKANA
PUTS:
        CMP     CX,Ø
        JE      PUTS_END
        MOV     AL,ES:[BX]
        INC     BX
        PUSH    BX
        PUSH    CX
        PUSH    ES
        CALL    PUTC
        POP     ES
        POP     CX
        POP     BX
        JC      QUIT
        LOOP    PUTS
PUTS_END:
        JMP     M_LOOP
;
;-- remain output buffer ? --
FLUSH:
        MOV     BX,SEG DATA3
        MOV     ES,BX
        CMP     WORD PTR ES:OUT_LEN,Ø
        JE      QUIT
        CALL    FLUSH_SUB
QUIT:
        EXIT
```

```
;-- get char --
GETC      PROC
          CMP       WORD PTR IN_LEN,Ø
          JE        GETC_1
          MOV       DI,IN_PTR
          MOV       AL,[DI]
          INC       WORD PTR IN_PTR
          DEC       WORD PTR IN_LEN
          CLC
          JMP       GETC_END
GETC_1:
          MOV       CX,BUFSIZ
          MOV       DX,OFFSET DGROUP:IN_BUF
          READ_STDIN ……………図5-13のマクロ
          JC        GETC_END
          OR        AX,AX
          STC
          JZ        GETC_END
          DEC       AX
          MOV       IN_LEN,AX
          MOV       AL,IN_BUF
          MOV       IN_PTR,OFFSET DGROUP:IN_BUF+1
          CLC
GETC_END:
          RET
GETC      ENDP

;-- put char --
PUTC      PROC
          MOV       BX,SEG DATA3  } Ⓐと同様
          MOV       ES,BX         }
          CMP       WORD PTR ES:OUT_LEN,BUFSIZ
          JE        PUTC_1
          INC       WORD PTR ES:OUT_LEN
          MOV       BX,ES:OUT_PTR
          MOV       ES:[BX],AL
          INC       WORD PTR ES:OUT_PTR
          CLC
          JMP       PUTC_END
PUTC_1:
          PUSH      AX
          CALL      FLUSH_SUB
          POP       BX
          JC        PUTC_END
          CMP       AX,ES:OUT_LEN
          STC
          JNE       PUTC_END
          MOV       ES:OUT_BUF,BL
          MOV       WORD PTR ES:OUT_LEN,1
          MOV       ES:OUT_PTR,OFFSET ES:OUT_BUF+1   ;
          CLC
PUTC_END:
```

```
        RET
PUTC    ENDP

;-- buffer flush --
FLUSH_SUB       PROC
        PUSH    DS
        MOV     BX,ES
        MOV     DS,BX
        MOV     CX,ES:OUT_LEN
        MOV     DX,OFFSET ES:OUT_BUF
        WRITE_STDOUT  ……………図5-13のマクロ
        POP     DS
        RET
FLUSH_SUB       ENDP

CODE    ENDS

DATA1   SEGMENT
;
DATA1   ENDS
```

ローマ字カナ変換ルーチンを別モジュールにしたので，セグメントDATA1の中身がなくなってしまった．セグメントグループの定義を変更せずにすむように，そのまま残してある

```
DATA2   SEGMENT
;
IN_LEN  DW      ?
IN_PTR  DW      ?
IN_BUF  DB      BUFSIZ DUP (?)
;
DATA2   ENDS

DATA3   SEGMENT
;
OUT_LEN DW      ?
OUT_PTR DW      ?
OUT_BUF DB      BUFSIZ DUP (?)
;
DATA3   ENDS

STACK   SEGMENT STACK
        DB      STACKSIZ DUP (?)
STACK   ENDS

        END START
```

リスト 5-3 ローマ字カナ変換モジュール ROMASUB.ASM

```
        INCLUDE MSDOS.H

        PUBLIC  ROMAKANA

CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE,ES:CODE

;-- roma kana convert --
ROMAKANA        PROC
        MOV     WORD PTR CS:CNV_LEN,0
        CMP     BYTE PTR CS:CHR1F,0
        JNE     SECONDCHR
        CALL    ISALPHA
        JE      FSTCHR
        JMP     SET_END
FSTCHR:
        AND     AL,5FH
        CALL    BOIN
        JNE     STORE
        ;
        MOV     AL,CS:HYOU2[BX]
        JMP     SET_END
STORE:
        MOV     BYTE PTR CS:CHR1F,1
        MOV     CS:CHR1,AL
        JMP     ROMAKANA_END
SECONDCHR:
        CALL    ISALPHA
        JZ      TRANSFER
        ;
        XCHG    AL,CS:CHR1
        ;
        CALL    SETCHR
        MOV     AL,CS:CHR1
        MOV     BYTE PTR CS:CHR1F,0
        JMP     SET_END
TRANSFER:
        AND     AL,5FH
        XCHG    AL,CS:CHR1
        ;
        MOV     SI,OFFSET HYOU3
        MOV     CX,9
SLOOP:
        CMP     CS:[SI],AL
        JE      SFOUND
        ADD     SI,6
        LOOP    SLOOP
        ;
        CALL    SETCHR
        MOV     AL,CS:CHR1
```

プロシージャ名ROMAKANAをパブリックな名前として宣言する．この宣言がなければ，プロシージャROMAKANAを他のモジュールから呼び出すことはできない

ローマ字カナ変換プロシージャROMAKANA

入力パラメータ：ALレジスタに変換したい文字を入れて呼び出す

処理　：CXレジスタに変換後の文字数，EX:BXレジスタに変換後の文字列のアドレスを格納して返す

このプログラムは基本的にはリスト 5-1のローマ字カナ変換処理部分と同じであるが，以下に示す変更を加えてある

- COMモデルでも動作するように，データをセグメントCODEに置いた．
- 6章で解説するデバイスドライバでも利用できるように，DSレジスタの内容が不定でも動作するようにした．すなわちデータラベルの参照は，CSレジスタによるセグメントオーバーライドプリフィックスを使った．
- 変換後の文字列のアドレスのうち，セグメントアドレスをESレジスタに入れて返すようにした

```
        CALL    BOIN
        JNE     ROMAKANA_END
        MOV     AL,CS:HYOU2[BX]
        MOV     BYTE PTR CS:CHR1F,Ø
        JMP     SET_END
;
SFOUND:
        XCHG    AL,CS:CHR1
        CALL    BOIN
        JNE     CHKN
        ;
        MOV     AL,CS:[SI+BX+1]
        MOV     BYTE PTR CS:CHR1F,Ø
        JMP     SET_END
CHKN:
        CMP     CS:CHR1,'N'
        JNE     NOTKANA
        CMP     AL,'N'
        JNE     NOTKANA
        ;
        MOV     AL,'ﾝ'
        MOV     BYTE PTR CS:CHR1F,Ø
        JMP     SET_END
NOTKANA:
        XCHG    AL,CS:CHR1
SET_END:
        CALL    SETCHR
ROMAKANA_END:
        MOV     CX,CS:CNV_LEN
        MOV     BX,OFFSET CNV_BUF
        MOV     AX,CS   } 変換後の文字列のセグメントアドレスを
        MOV     ES,AX   } ESレジスタに入れて返す.
        RET
ROMAKANA        ENDP

;-- set char --
SETCHR  PROC
        MOV     BX,CS:CNV_LEN
        MOV     SI,OFFSET CNV_BUF
        MOV     CS:[SI+BX],AL
        INC     WORD PTR CS:CNV_LEN
        RET
SETCHR  ENDP

;-- boin --
BOIN    PROC
        MOV     BX,Ø
BLOOP_1:
        CMP     CS:HYOU1[BX],AL
        JE      BFOUND_1
        INC     BX
        CMP     BX,5
```

```
        JBE     BLOOP_1
BFOUND_1:
        RET
BOIN    ENDP

;-- isalpha --
ISALPHA PROC
        CMP     AL,'A'
        JB      NOTALPHA_2
        CMP     AL,'Z'
        JBE     ALPHA_2
        CMP     AL,'a'
        JB      NOTALPHA_2
        CMP     AL,'z'
        JA      NOTALPHA_2
ALPHA_2:
        CMP     AL,AL
NOTALPHA_2:
        RET
ISALPHA ENDP

;-- work area --
CHR1F   DB      Ø
CHR1    DB      Ø
CNV_LEN DW      ?
CNV_BUF DB      2 DUP (?)

;-- conversion table --
HYOU1   DB      'A','I','U','E','O'
HYOU2   DB      'ｱ','ｲ','ｳ','ｴ','ｵ'
;
HYOU3   DB      'K','ｶ','ｷ','ｸ','ｹ','ｺ'
        DB      'S','ｻ','ｼ','ｽ','ｾ','ｿ'
        DB      'T','ﾀ','ﾁ','ﾂ','ﾃ','ﾄ'
        DB      'N','ﾅ','ﾆ','ﾇ','ﾈ','ﾉ'
        DB      'H','ﾊ','ﾋ','ﾌ','ﾍ','ﾎ'
        DB      'M','ﾏ','ﾐ','ﾑ','ﾒ','ﾓ'
        DB      'Y','ﾔ','ｲ','ﾕ','ｴ','ﾖ'
        DB      'R','ﾗ','ﾘ','ﾙ','ﾚ','ﾛ'
        DB      'W','ﾜ','ｲ','ｳ','ｴ','ｦ'

CODE    ENDS
;
        END
```

アセンブルの手順は，次に示すようにソースファイルが１つしかない場合とまったく同様です．アセンブルするとそれぞれのソースファイルごとにオブジェクトファイルが作成されます．

```
MASM  ROMA;
MASM  ROMASUB;
```

これに対し，リンクの操作はこれまでとは少し異なります．リンクの操作では，2つのオブジェクトファイル「ROMA.OBJ」と「ROMASUB.OBJ」を結合して1つの実行ファイル「ROMA.EXE」を生成します．そのためには次のようにオブジェクトファイルの名前を2つとも指定しなければなりません．

```
LINK  ROMA+ROMASUB;
```

オブジェクトファイルの名前はこのように＋(プラス)記号で区切って並べます*．生成される実行ファイルの名前は，先頭に指定したオブジェクトファイルと同じものになります．すなわちこの場合，「ROMA.EXE」となります．

分割アセンブルの仕組み

以上に示したように，分割アセンブルの手順はこれまでのアセンブル方法と比べてそれほど難しいものではありません．わずかに余分な操作が必要になるだけで，それによって受けるメリットに比べればたいしたことはありません．分割アセンブルの仕組みを理解しておくと，こうした余分な操作や擬似命令の意味がよくわかります．そこで，分割アセンブルがどのような仕組みで実現されるのかを解説しましょう．

ソースファイルをアセンブルすることによって生成されるオブジェクトファイルには，マシン語コードやプログラム中で定義したデータが含まれています．ただし，ラベルに対応するアドレス値にはモジュール先頭からのアドレスが「仮のアドレス」として割り当てられています．さらに，各モジュールにおいてパブリック宣言されたプロシージャ名について，その名前と型属性，およびモジュール先頭からのアドレスなどの情報が，マシン語コードに加えてオブジェクトファイル中に格納されています．

---

*＋記号の代わりにスペースでもよい．＋の方が結合するという意味をはっきり表しているので＋記号を用いた．

オブジェクトファイルを結合して実行ファイルを作成するリンクの操作では，同じ名前のセグメントを結合しながら，それまで仮に割り当てられていたアドレス値を実際にセグメント内に配置したアドレスに置き換えます．そして，パブリックなプロシージャを呼び出している部分を，確定したプロシージャのアドレスで置き換えます．この操作を**外部参照の解決**と呼びます．

この仕組みを図5-20に示します．このようにオブジェクトファイルにはアセンブル時に割り当てられる仮のアドレスに加え，リンク時にあらためてアドレスを割り当てるためのさまざまな情報が格納されているので，**リロケータブルオブジェクト**（再配置可能なオブジェクト）と呼ばれます．

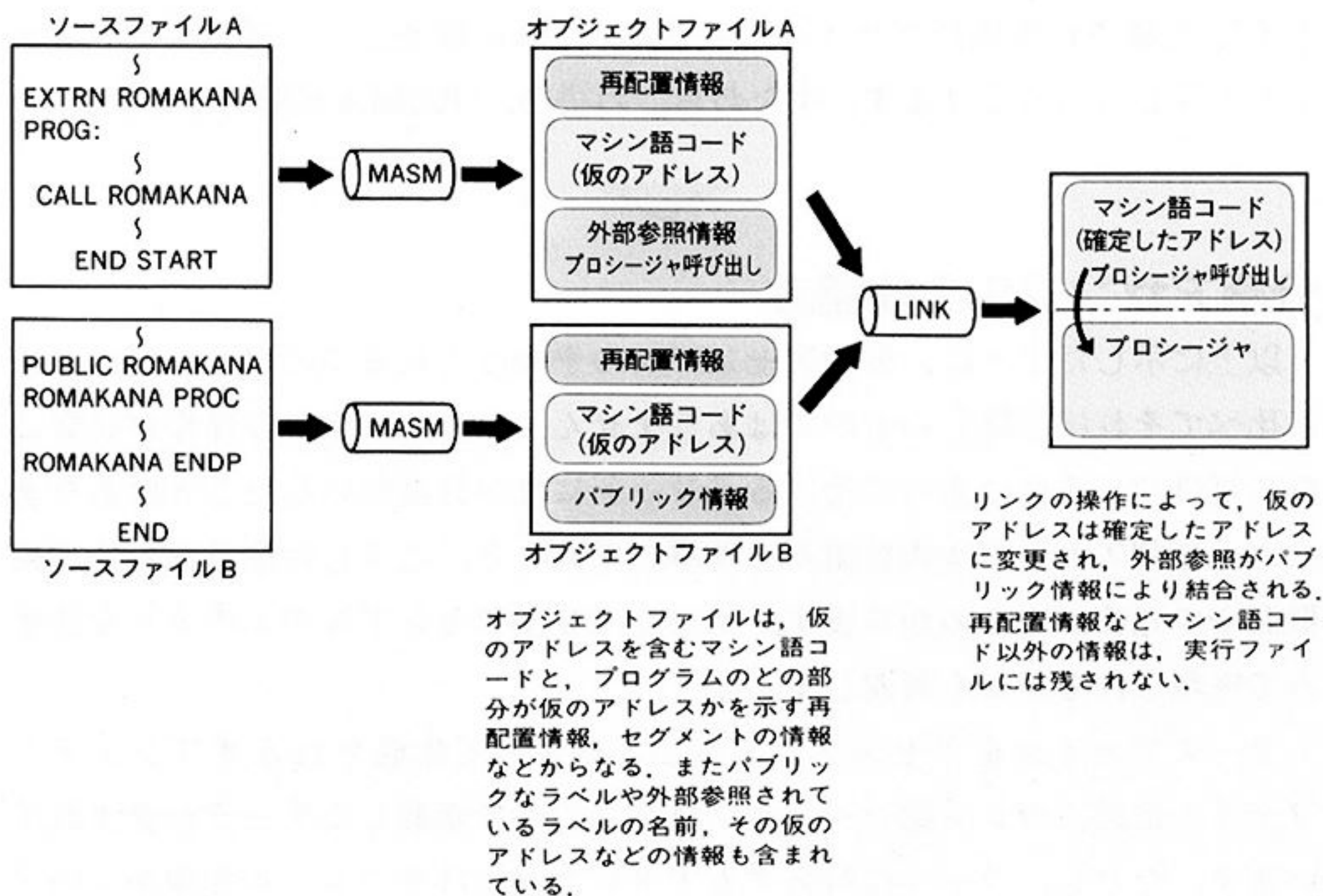



図5-20　分割アセンブルの仕組み

なお，リンクの操作で確定するアドレスはセグメント内のオフセットアドレスだけで，セグメントアドレスは4章で解説したように仮のアドレスが割り当てられたままとなります．セグメントアドレスは実行時に，MS-DOSによりメモリ上にロードされる際に初めて確定します．

リンカのライブラリ検索機能

いろいろなプログラムで部品として利用できるモジュールをいくつも作成すれば，新しいプログラムを作るときに必要なものだけをリンクして利用することができます．MASMを中心とするプログラミングツールには，このサブルーチン群を効率的に活用する実に便利な機能が用意されています．それは「ライブラリ」を扱う機能です．

ライブラリはモジュール群を1つのファイルにしたものです．ライブラリの作成方法はあとで示しますが，簡単な操作でいくつものモジュールのオブジェクトを1つのファイルにまとめてしまうことができます(図5-21)．

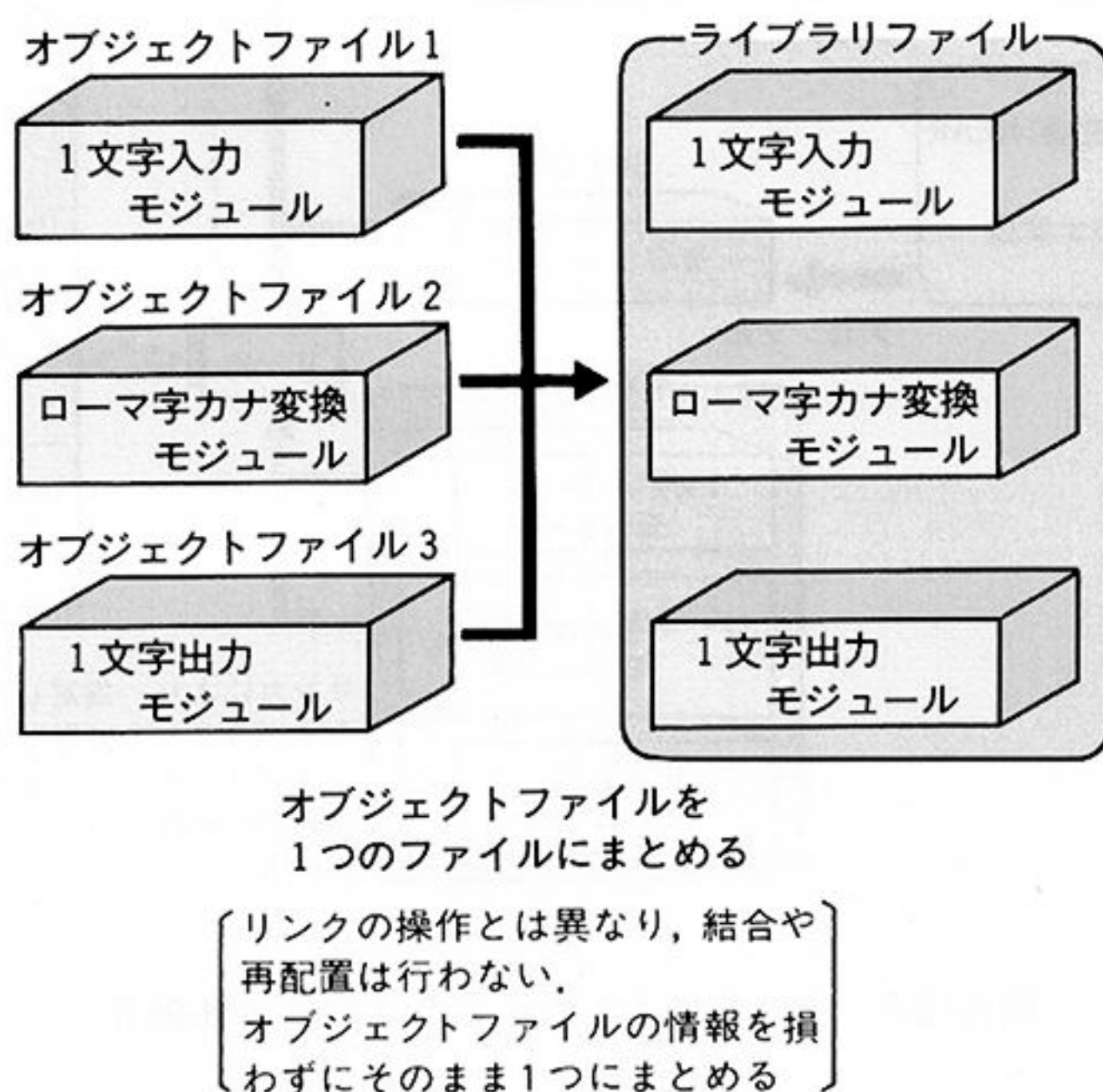



図5-21 ライブラリとは

そして便利な機能とは，このライブラリのなかから必要なモジュールを抜き出して自動的にリンクするというリンカ(LINKコマンド)の機能です．

たとえば，メインモジュールやその他のモジュールでライブラリ中にあるモジュールのプロシージャが呼び出されているとします．リンク時にライブ

ラリ名を指定すると，リンカはそのライブラリのなかから必要なプロシージャを含むモジュールを探してリンクします．ユーザー自身がオブジェクトファイルのリストとしてモジュール名をいくつも並べる必要はないのです．リンカは，指定したオブジェクトモジュールを必ずリンクしますが，そのなかに含まれないプロシージャが呼び出されているとライブラリを自動的に検索して必要なモジュールを見付けたらそれを抜き出してリンクします(図 5-22)．

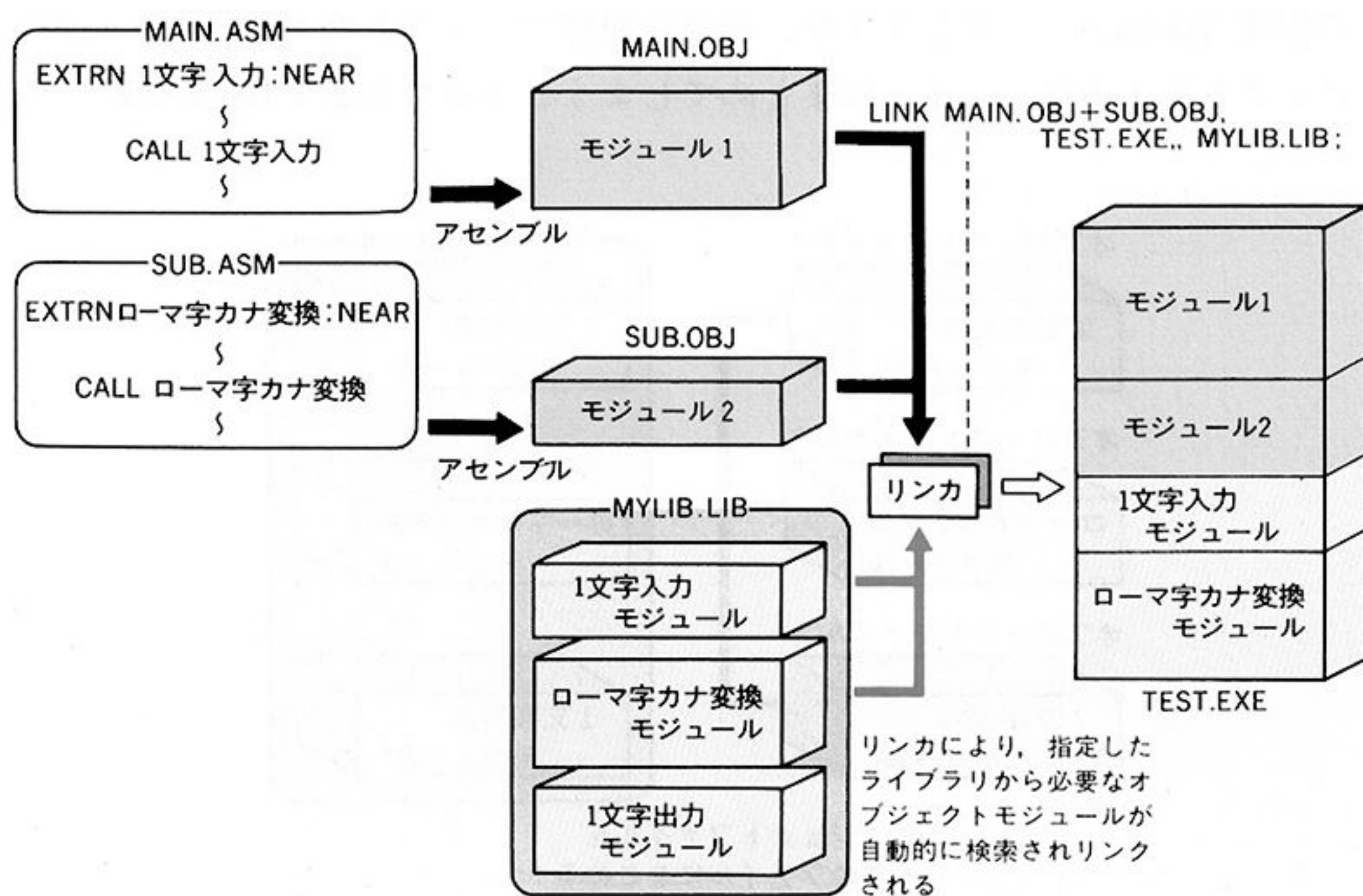



図 5-22　リンカによるライブラリ自動検索機能

リンカにライブラリの名前を指示するには次のようにします．これは，MAIN，A，Bという3つのモジュールとMYLIBというライブラリをリンクする例です．くわしくはAPPENDIXを参照してください．

```
LINK  MAIN+A+B, , , MYLIB
```

ライブラリアンの働き

ライブラリを作成し，保守するためのツールがライブラリアン(LIB コマンド)です．ライブラリアンは図 5-23 のようにライブラリを作成し，管理する機能を持ちます．

先に述べたように，いくつかのオブジェクトファイルをライブラリアンによって1つにまとめ，ライブラリを作成します．そしてライブラリアンによって，ライブラリにオブジェクトモジュールを追加したり，ライブラリからオブジェクトモジュールを抜き出すなどの編集を行い，ユーザー独自のライブラリを作ることが可能です．

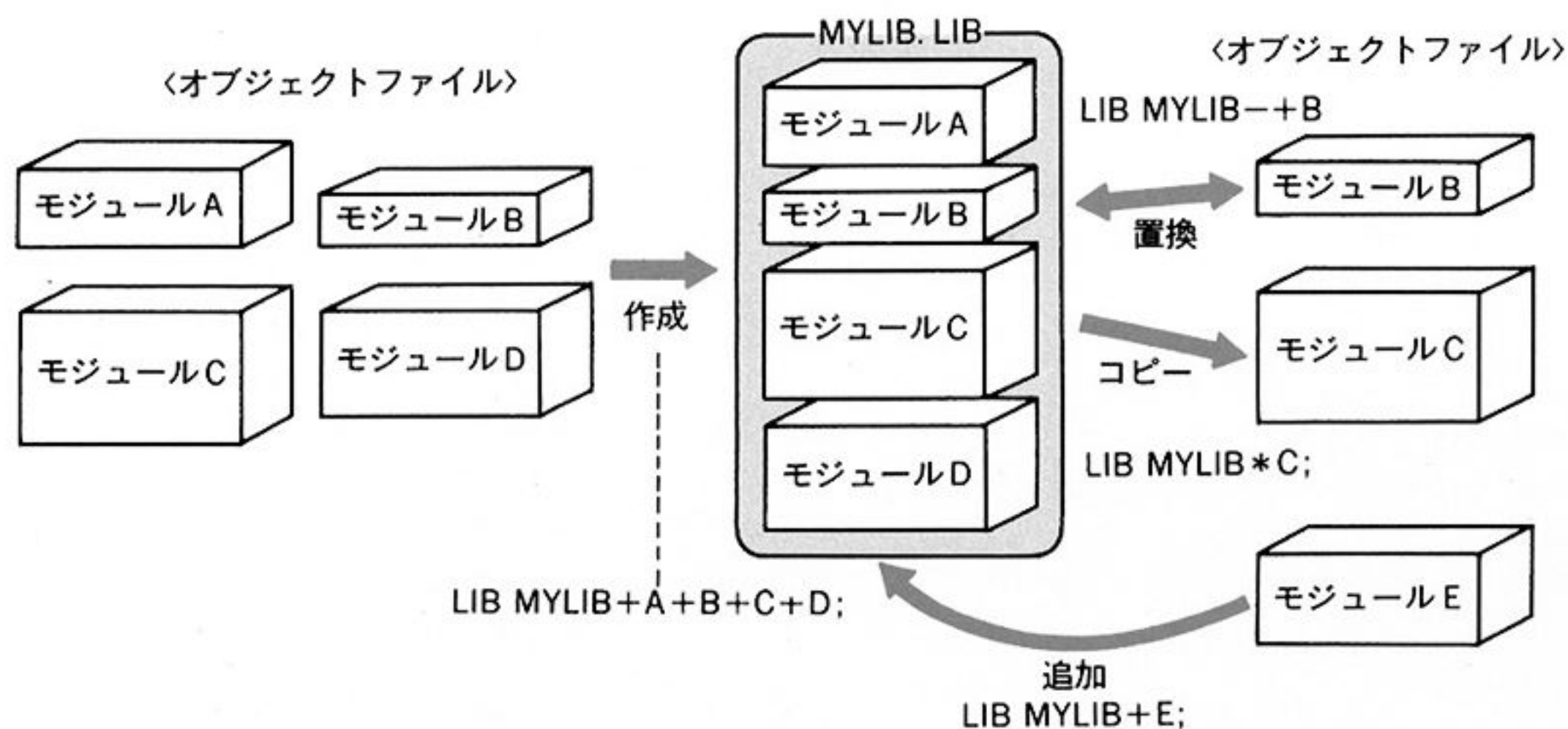
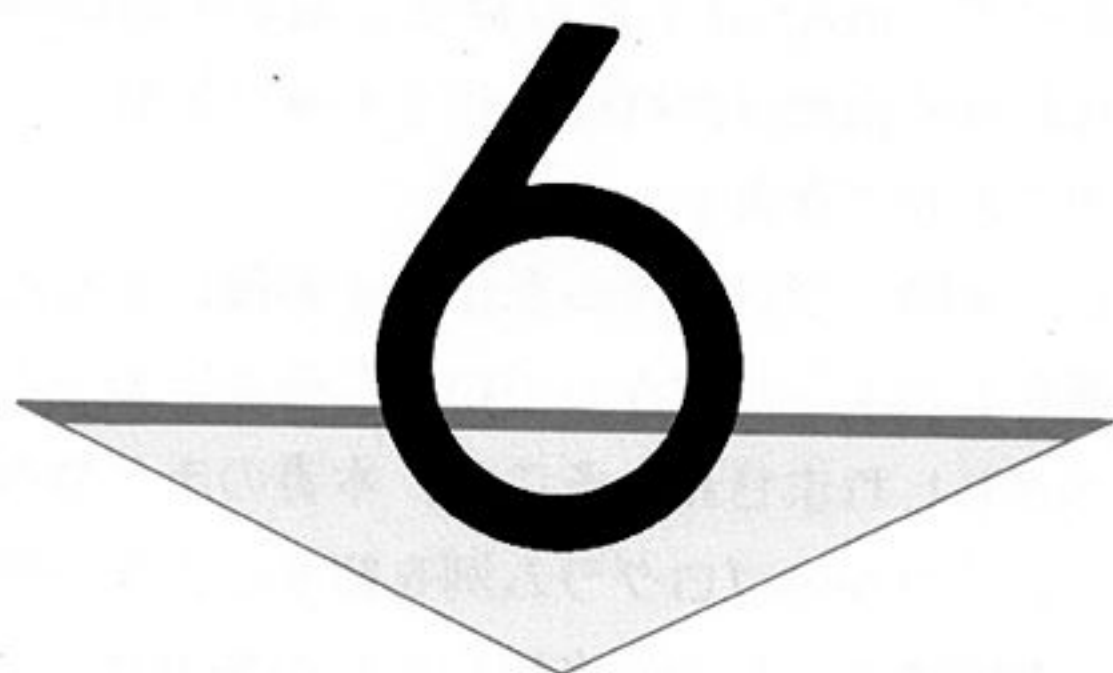



図 5-23　ライブラリアンの働き

ライブラリアンのくわしい操作方法については，APPENDIX を参照してください．

アセンブラ
実用テクニック

前章までで，MASM の擬似命令に関する話は終わりです．これだけの知識があれば，もう十分にアセンブラを使いこなすことができます．

しかし，実際にプログラムを作成する段になると，これらの知識をどのように生かしていったらよいのかとまどってしまうかもしれません．そこで，本書のまとめの意味も含めて，いくつかのプログラム例を紹介します．例題は，なるべく興味深く，しかも実用的なものを選びました．本書のプログラムが，アセンブラを自在に活用するための足掛かりになることを期待します．

6.1

C言語とのリンク

市販のアプリケーションプログラムのなかで，アセンブラだけで書かれたものはほとんどありません．今後ますますその傾向は強くなると思われます．といっても，アセンブラをまったく使わないわけではありません．逆にいえば，アセンブラが使われないことはほとんどないといってもよいでしょう．

2.3章で解説したのでおわかりだと思いますが，プログラムをC言語などの高級言語で記述するにしても，必要に応じてプログラムの一部をアセンブラで記述する場合が多いのです．したがってアセンブラと他の言語をリンクするテクニックは，非常に重要なものとなっています．

本節では，オセロゲームの思考ルーチンを題材として，MS-DOSで最もポピュラーなプログラミング言語の1つであるC言語とアセンブラ(MASM)をリンクする方法を解説します．

アセンブルとコンパイル

アセンブリ言語のプログラムは「アセンブル&リンク」によって実行ファイルを生成します．これに対し，C言語のプログラムは「コンパイル&リンク」によって実行ファイルを生成します．コンパイルという操作はC言語で書かれたプログラムをマシン語に変換するという操作です．アセンブラがアセンブリ言語のニーモニックを1対1に対応するマシン語命令に置き換えていくのに対し，コンパイラはC言語の構文ごとに一定のマシン語命令の組み合わせに置き換えていくのです．

コンパイラの生成するオブジェクトファイルは，アセンブラの生成するオブジェクトファイルとなんら変わりはありません*．したがってアセンブリ言

---

*処理系によってはMS-DOS標準のオブジェクトファイルではなく，独自形式のオブジェクトファイルを生成するものもある．この場合，アセンブラもMASMではなく独自のアセンブラでなければリンクできない．

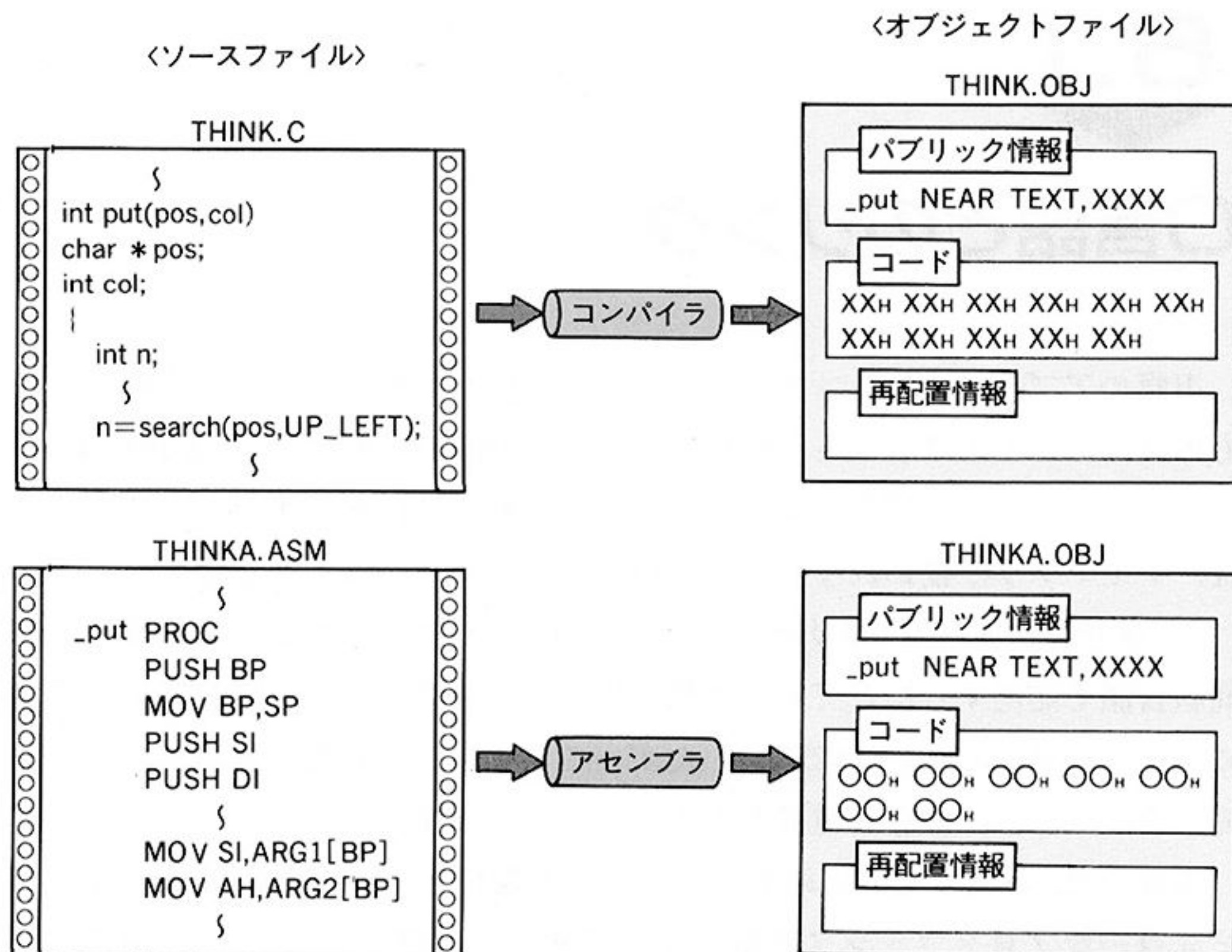



図 6-1　コンパイラとアセンブラ

語のモジュールとして作成したプログラムのオブジェクトファイルと，そのままリンクすることができます．基本的には，他のプログラムとリンクするモジュールを作成するのと同じ考え方でプログラムを作成すれば，それでよいのです(図6-1)．

ただし，いくつかの点においてプログラムの形式を一致させる必要があります．その具体的な形式については，以下の節で解説します．

関数＝PROC

C言語ではプログラムを「関数」という単位で作成します．ここでは関数を「サブルーチン」のことだと思ってかまいません．MASM ではサブルーチンのことを「プロシージャ」と呼びます．C言語の関数と MASM のプロシージャは，名前が異なるだけで実体は同じものです．

C言語とMASMのプログラムをリンクするには，MASM側でサブルーチンをプロシージャとして作成します．もちろん以下に解説するように，各処理系ごとの約束にしたがって各種の設定をしなければなりませんが，基本的にはプロシージャとして定義すればよいのです．

5章で解説したように，プロシージャはPROC擬似命令で定義します．そしてプロシージャを他のモジュール(C言語の関数)から呼び出せるようにPUBLIC宣言します．これが基本です．具体的な定義例を図6-2に示します．

〈C言語の関数〉

```
int     put(pos,col)
char     *pos;
int     col;
{
            ∫
        プログラム
            ∫
}
```

〈アセンブラのプロシージャ〉

```
_put PROC
    ∫
プログラム
    ∫
_put ENDP
```

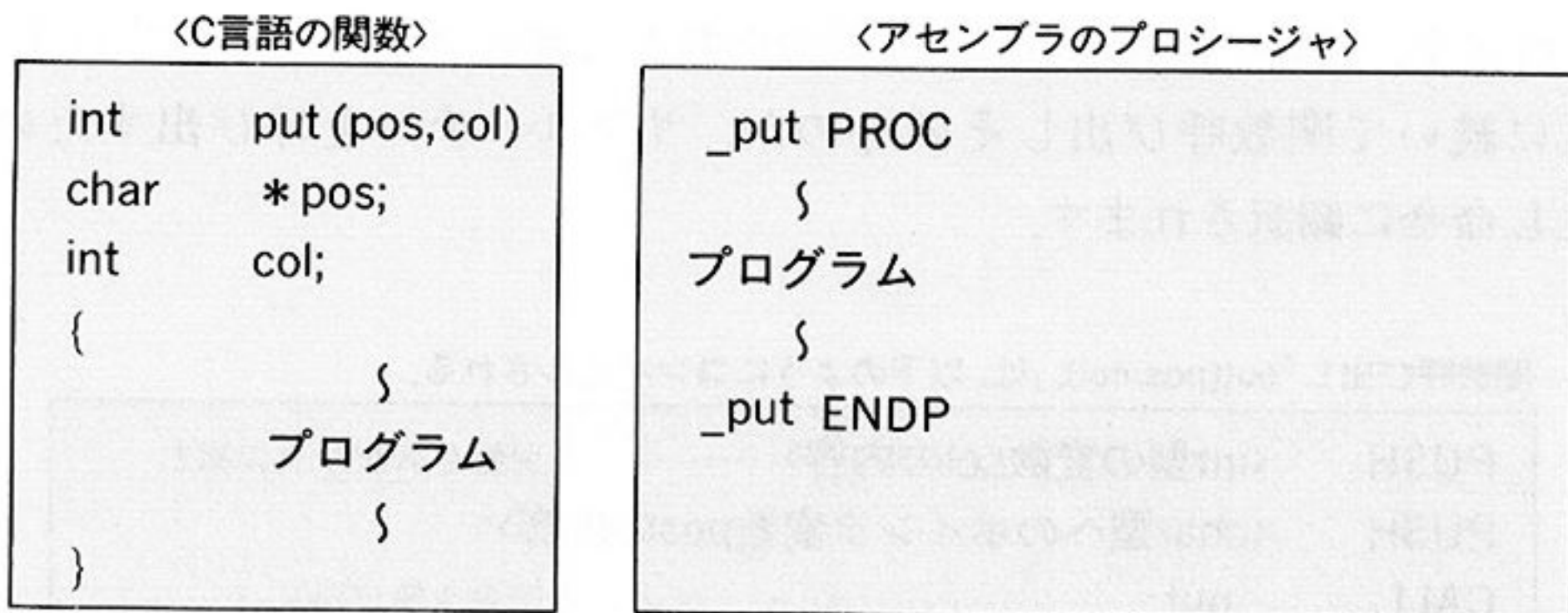



図6-2　C言語の関数とアセンブラのプロシージャの記述

C言語からMASMへの引数の受け渡し

C言語の関数はパラメータとして「引数」を指定することができます．たとえば，

```
putchar(c);
```

という関数呼び出しでは，「変数cの値」を引数として関数putchar()を呼び出しています*．

```
printf("%d¥n", i);
```

では，「"%d¥n"という文字列へのポインタ」，および「変数iの値」の2つを引数として関数printf()を呼び出しています．

*処理系により異なるが，通常はputchar()は関数ではなくマクロであり，putc()関数に置き換えられる．

MASMのプロシージャにパラメータを渡したい場合も，同じように関数の引数として指定するのですが，それをどうやって受け取ればよいのでしょうか．それを解説するために，C言語の関数呼び出しの手順をまず説明しましょう．CコンパイラはC言語で書かれたプログラムをマシン語プログラムに翻訳しますが，どのようなマシン語に翻訳されるかがわかれば，引数受け渡しの仕組みがわかります．

関数の呼び出しは，図6-3に示すようなマシン語プログラムに翻訳されます．まず，関数に引数がある場合は，PUSH命令によって引数がスタックに積まれます．引数が複数あれば，右側の引数から順にスタックに積まれます．それに続いて関数呼び出しそのものが，サブルーチンを呼び出すためのCALL命令に翻訳されます．

関数呼び出し「put(pos,col);」は，以下のようにコンパイルされる．

```
PUSH    <int型の変数colの内容>…………引数をスタックに積む
PUSH    <char型へのポインタ変数posの内容>
CALL    _put……………………………………関数を呼び出す
ADD     sp,4 ……………………………………スタックレベルをもとに戻す
```

図6-3　関数呼び出しの手順

この図6-3から，関数への引数はスタック領域を通じて受け渡されることがわかります．

今度は逆に，呼び出される関数の方はどのように翻訳されるのかを見てみましょう．次の図6-4に示すように，関数の入り口ではまず，BPレジスタをスタックにプッシュします．そしてSPレジスタ(スタックポインタ)の内容をBPレジスタにロードします．さらに，関数内で定義された変数領域として必要なバイト数分だけSPレジスタの内容を減らします．

```
_put    PROC
        PUSH    BP
        MOV     BP,SP
        SUB     SP,××H
```

図6-4　関数の入口の手順

ここで4.6章のローカル変数についての解説(特に149ページの図4-29)を思い出してください．図6-4の手順はローカル変数を確保する手順とよく似ています．実はC言語の関数内で定義された変数は，ローカル変数として確保されているのです*．

ローカル変数はスタック上に一時的にデータ領域を確保し，必要がなくなったら開放するというものでした．ローカル変数をアクセスするためには，BPレジスタをポインタとして利用します．BPレジスタをポインタとするアドレッシングモードではSSレジスタの指すセグメントがデータセグメントとして扱われるからです．

呼び出した側の関数では，引数をスタックに積んでから呼び出しを行います．したがってスタックは図6-5のような状態になっています．関数を呼び出すためのCALL命令により戻り先のアドレスがスタックに積まれており，さらにBPの値がスタックに積まれていることに注意すれば，図6-5に示すようなアドレッシングで引数にアクセスできることがわかります．引数も呼び出す側で確保してくれた一種のローカル変数としてアクセスできるわけです．

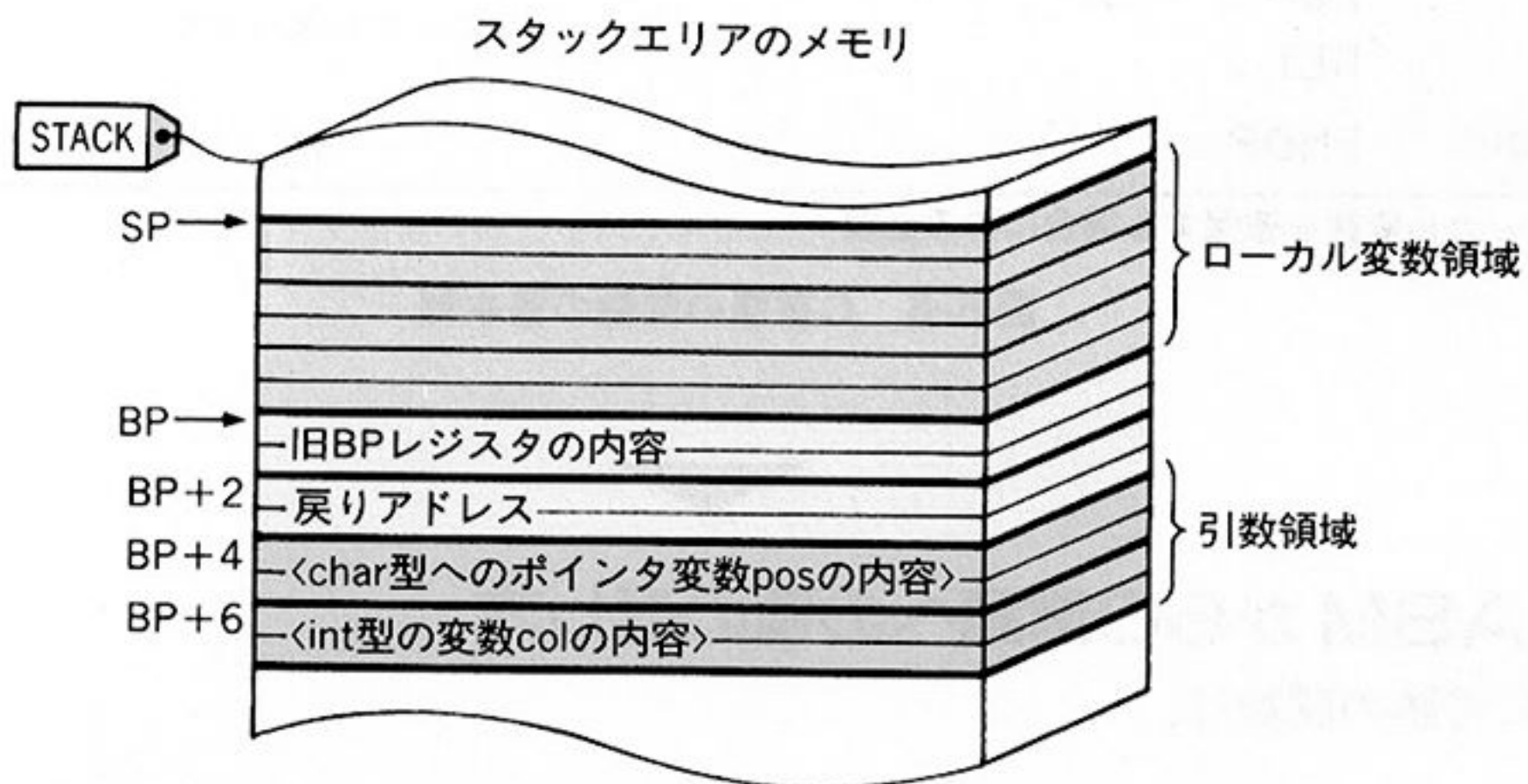



図6-5 スタックに積まれた引数

*関数外で定義された変数や，関数内で定義されてもstatic宣言された変数は，データセグメントの固定された領域に確保される．

BPレジスタはローカル変数を扱うため，そしてスタックを通してパラメータを受け渡しするために用意されているレジスタです．このような使い方をするレジスタは一般に「フレームポインタ」とも呼ばれます．

関数を抜ける前，つまりRET命令の前にはSPレジスタからローカル変数として確保したバイト数を加え，BPレジスタをPOPしてもとの値に戻しておかねばなりません．これらのことを総合すると，C言語から呼び出せるプロシージャは図6-6に示すような構造を基本形にすればよいことがわかります．

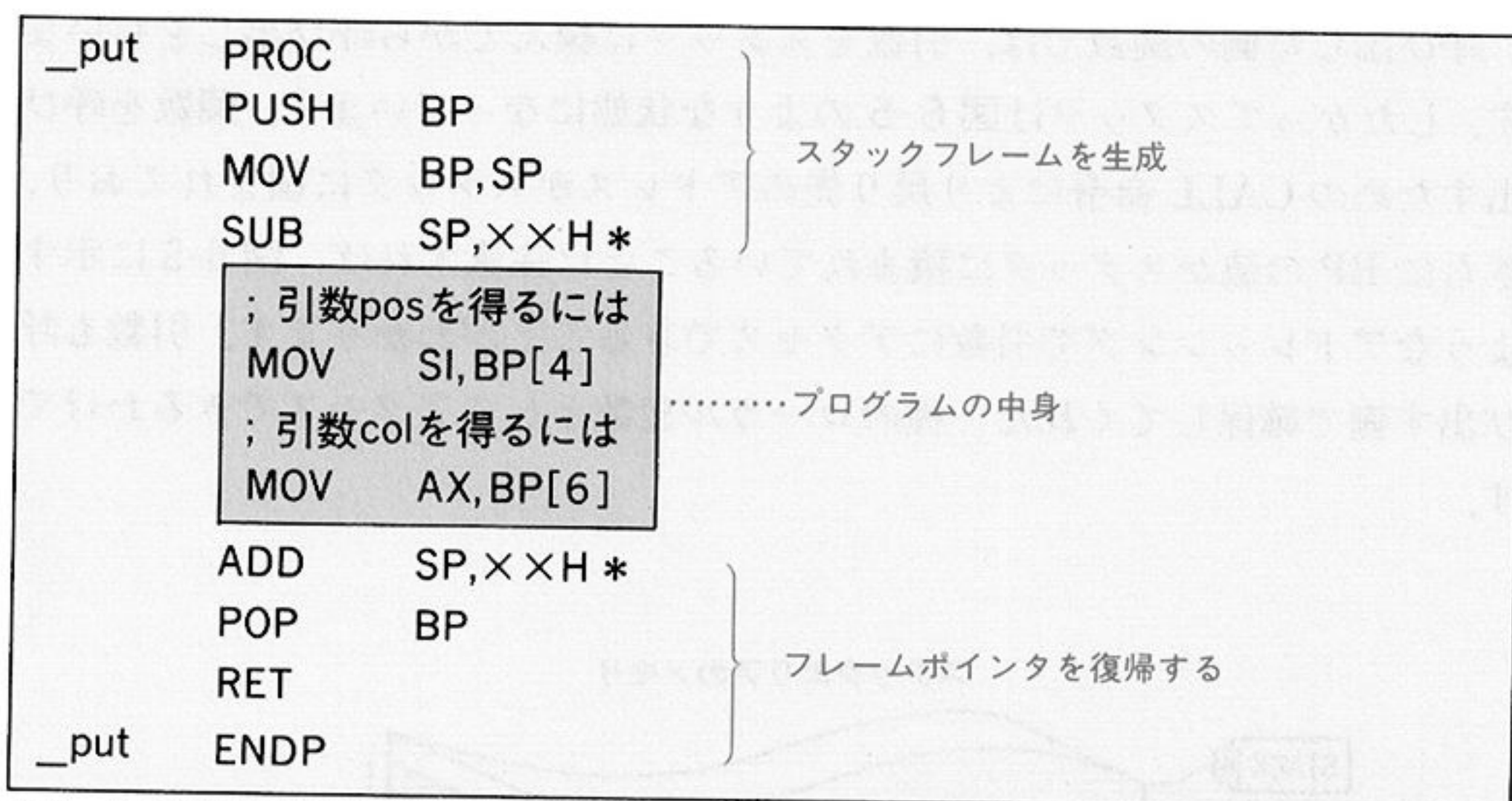



図6-6　C言語の関数の基本形

MASMからC言語への値の返し方

C言語の関数は，

```
return c;
```

のように，呼び出した関数に値を返すことができます．この仕組みも解説する必要があるでしょう．答えは簡単で，値をあるレジスタに格納してRETすればよいのです．

どのレジスタを使って値を返すかは，関数の型，つまり返す値の型によって違います．一般には，**表6-1**のようなレジスタを使って値を返します．ただし，必ずしもこの通りではない場合もありますので，使用しているコンパイラのマニュアルをよく読んでみてください．

返す値の型	MS-C/Turbo C	Lattice C
(unsigned)char	AX	AL
(unsigned)short	AX	AX
(unsigned)int	AX	AX
(unsigned)long	上位ワードはDX 下位ワードはAX	上位ワードはBX 下位ワードはAX
near ポインタ	AX	AX*
far ポインタ	セグメントアドレスはDX オフセットアドレスはAX	セグメントアドレスはBX オフセットアドレスはAX*

＊関数へのポインタの場合は異なる

表6-1 関数値の返し方

スモールモデルとラージモデル

8086系のCPUを使っている限り避けて通れないのがメモリモデルの問題です．MS-DOSの実行型ファイルには2つの形式があることは先に何度か述べました．1つは，プログラムがたった1つのセグメントからなる「COMモデル」で，もう1つは複数のセグメントからなる「EXEモデル」です．この両者は，場合に応じて使いわけることを4.8章でくわしく解説しました．

C言語で作成するプログラムも同じように2種類に分けられます．COMモデルに対応する**スモールモデル**（ただし実行ファイルはEXE型）と，EXEモデルに対応する**ラージモデル**です．スモールモデルではポインタとしてオフセットアドレスのみが使用されるので，比較的高速な処理が可能です．ラージモデルはポインタとしてさらにセグメントアドレスも使用されるので，処理は遅くなりますが多くのデータを扱うことができます．

スモールモデル，ラージモデルの違いをまとめたのが図6-7です．なお，スモールモデルのプログラムは1つのセグメントだけから構成されているわけではなく，コード部分とそれ以外でそれぞれ1つのセグメントを構成するので，実行型ファイルとしてはEXEモデルになります*．

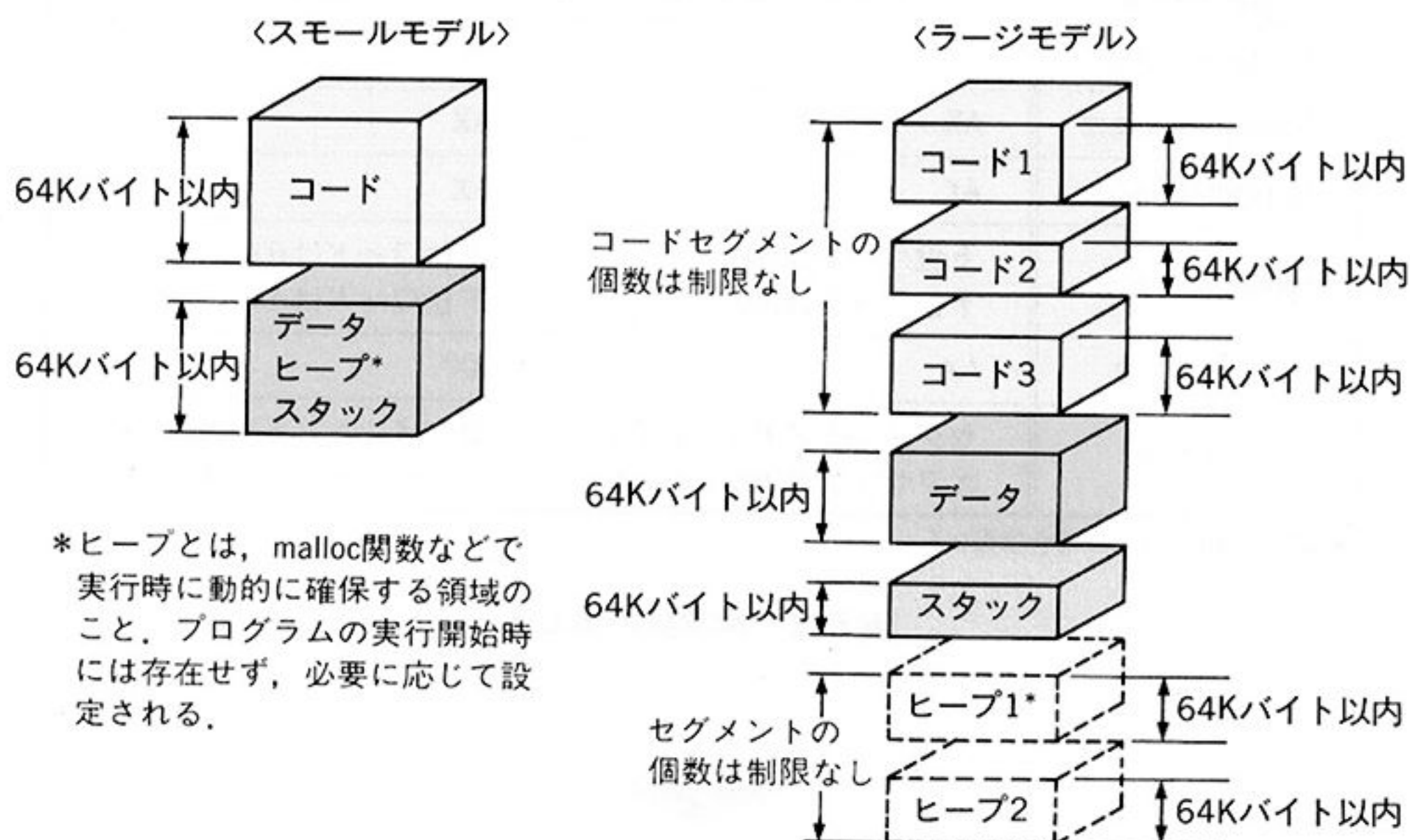



図6-7　スモールモデルとラージモデル

実は今までは，スモールモデルを対象に，C言語とリンクするためのMASMのプログラムについて解説してきました．ラージモデルの場合は，以下の点が異なります．

まず，プロシージャはFARタイプとして定義しなければなりません．ラージモデルでは，どの関数もかならず他のセグメントにあるとしてファーコールで呼び出されるからです**．具体的には図6-8のように定義します．

---

* Turbo-Cなど処理系によっては，COMモデルのプログラムを作成することもできる．この場合は4章で解説したように，実行ファイルの大きさが小さくなりロードも速くなるというメリットがある．

**ラージモデルでは同じセグメント内にある関数でもファーコールで呼び出される．したがってどのセグメントから呼び出されるかに関係なくFAR型として定義すればよい．

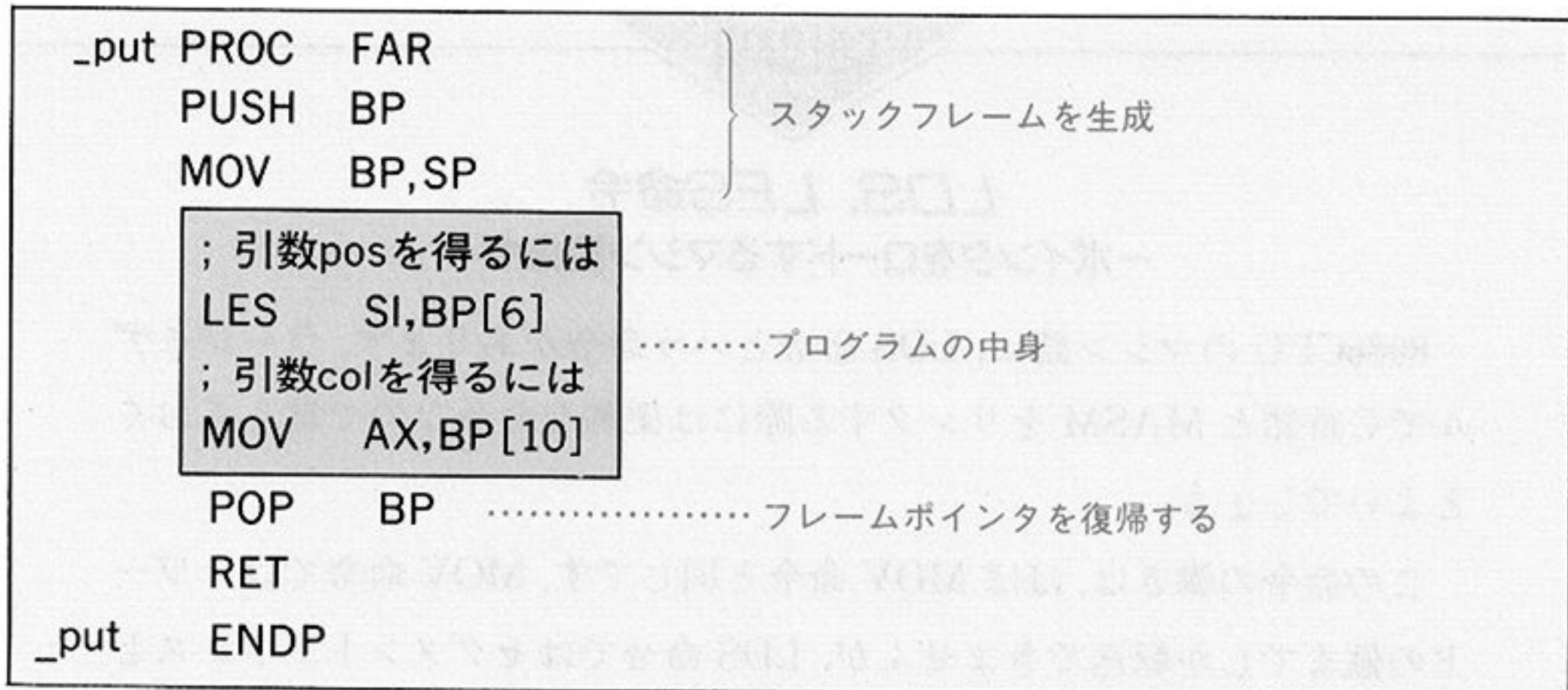



図 6-8 FAR 型のプロシージャの定義

ラージモデルでは関数の呼び出しがセグメント外CALLなので，スタックの状態も異なります．図 6-9 に示すように CALL 命令で積まれる戻りアドレスにセグメントアドレスが加わります．また，引数がポインタの場合もセグメントを含めた 2 ワードがスタックに積まれるので，注意が必要です*.

図 6-9 ラージモデルの関数におけるスタックの状態

*スモールモデルとラージモデルではプログラムを一部変更しなければならない．ここで 5.1 章で解説した IF〜ELSE〜ENDIF 擬似命令（190ページ参照）による条件アセンブルを利用すれば，わずかの変更でどちらのモデルにも対応するプログラムを書くことができる．

COLUMN

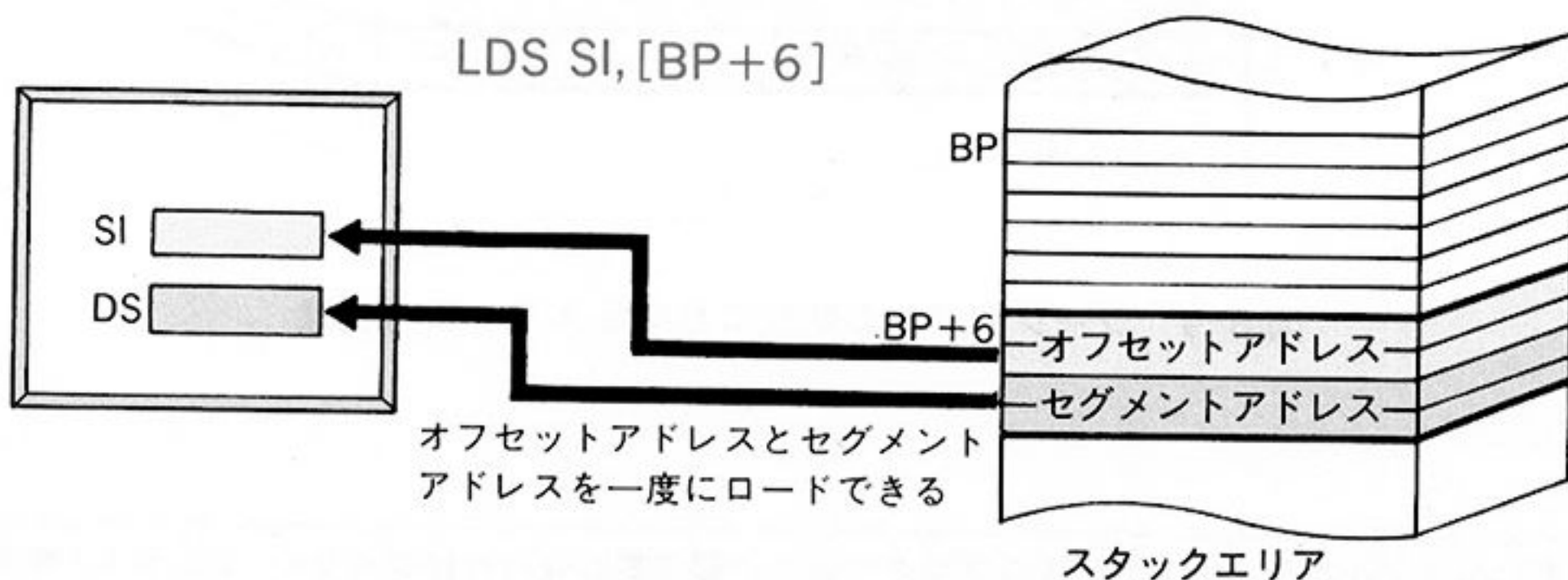
LDS, LES命令
―ポインタをロードするマシン語命令―

8086CPUのマシン語に, LDS命令という命令があります. ラージモデルでC言語とMASMをリンクする際には便利な命令なので覚えておくとよいでしょう.

この命令の働きは, ほぼMOV命令と同じです. MOV命令では1ワードの値までしか転送できませんが, LDS命令ではセグメントアドレスとオフセットアドレスからなる2ワードのポインタ値を, メモリからレジスタにいっぺんにロードすることが可能です. したがって, ラージモデルの関数で引数として渡されたポインタをロードするために利用することができます. 具体的には,

```
LDS   SI,  [BP+6]
```

のように使います. この場合, [BP+6]のメモリの内容をSIレジスタに転送します. さらに, それに続くメモリの内容をDSレジスタに転送します. その値がポインタであることを考えれば, ポインタ値のオフセットアドレスをSIレジスタに, セグメントアドレスをDSレジスタにロードしたことになります(図を参照).

同様にLES命令では, セグメントアドレス部分をESレジスタにロードします.

MASMとC言語をリンクするための約束事

これまで解説してきたC言語とリンクするための約束事は，ほとんどの処理系に共通したものです．これ以外にも処理系によってさまざまな約束事があり，アセンブリ言語でC言語の関数を記述する際には必ず守らなければなりません．

ここでは一般的な約束事を挙げておきます．くわしくは使っている処理系のマニュアルを調べてください．

セグメント名を一致させる

特にスモールモデルでは，セグメントの名前はコンパイラが出力するものと同じにしておかなければなりません．ラージモデルでは，独立したセグメントを確保するために別な名前でセグメントを定義するべきですが，それほど大きなモジュールでなければスモールモデルの場合と同じ名前にしておけばよいでしょう．

レジスタ，フラグの内容を保存する

セグメントレジスタやベースポインタ(BPレジスタ)の内容はかならず保存しておかなければなりません*．そのほかにも処理系によっては保存しなければならないレジスタがあります．それらのレジスタをプログラムのなかで使用した場合は，プロシージャを抜ける前に呼び出された時点での値に戻しておかねばなりません．

たとえば，Microsoft C(MS-C)では前述のレジスタに加え，SIおよびDIレジスタを保存する必要があります**．さらにフラグレジスタ(ディレクションフラグ)も保存しなければなりません．

---

*ただし，処理系によってはセグメントレジスタのうちESレジスタは保存しなくてもよい．
**レジスタ変数として使用されているため．

関数名

C言語で呼び出す際に定義する名前と，MASMにおける名前がまったく同じであるとは限りません．たとえばMS-Cの場合，C言語で定義した名前の先頭に「_」(アンダースコア)を付けたものがMASMにおける名前になります．コンパイラが「_」を付けた関数名をパブリックなラベルとしてオブジェクトファイルに出力してしまうのです．

また，MASMではラベル名などに小文字を使ってもすべて大文字に変換されてしまいます．これに対し，C言語の関数名は小文字で定義するのが普通です．LINKコマンドは大文字と小文字を区別しないので特に問題はないのですが，コンパイラがLINKコマンドを自動的に呼び出す場合には，大文字と小文字を区別するオプションを指定して呼び出している処理系もあります*．その場合は，うまくリンクできません．そこで，MASMでも大文字と小文字を区別するようにオプションを指定してアセンブルします**．

コンパイラによるアセンブラソースの出力

C言語処理系の多くは，コンパイル後のオブジェクトプログラムをアセンブリ言語のソースプログラムとして出力できるようになっています．この機能を利用すれば，比較的簡単にC言語とリンクできるアセンブリ言語のサブルーチンを作成することができます．

まず，MASMで記述したい関数をC言語のソースプログラムとして書きます．プログラムの本体は必要ではなく，関数の名前と引数だけを定義すれば十分です．このソースファイルを，アセンブラソースを出力するオプションを付けてコンパイルします．出力されたプログラムはC言語から呼び出せるように記述されており，しかも入口および出口の処理も記述されています．目的のプログラムは，その間に挿入すればよいのです．

図6-10にMS-Cの場合を示します．他の処理系についてはそれぞれのマニュアルを参照して，各自で実際にやってみてください．

---

*/NOIGNORE オプション．くわしくはAPPENDIX参照のこと．

**/ML オプション．具体的な使用法は，269ページ図6-12のアセンブル実行例やAPPENDIXを参照のこと．

```
A>TYPE PUT.C ↵
int put(pos,col)     関数の名前と引数を定義する
 char *pos;
 int  col;
{} ………………………プログラムの中身は空でよい

A>MSC -Fa PUT; ↵ ……………アセンブラ・ソースファイルを出力(MS-Cの場合)
Microsoft (R) C Compiler  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985, 1986.  All rights reserved.

A>TYPE PUT.ASM ↵ ……………アセンブラ・ソースファイルの中味を表示する
;        Static Name Aliases
;
         TITLE    PUT
;        NAME     PUT.C

         .287
_TEXT    SEGMENT  BYTE PUBLIC 'CODE'
_TEXT    ENDS
_DATA    SEGMENT  WORD PUBLIC 'DATA'
_DATA    ENDS
CONST    SEGMENT  WORD PUBLIC 'CONST'                  セグメントの設定が
CONST    ENDS                                          行われている
_BSS     SEGMENT  WORD PUBLIC 'BSS'
_BSS     ENDS
DGROUP   GROUP    CONST,  _BSS,   _DATA
         ASSUME   CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
EXTRN    __chkstk:NEAR
_TEXT       SEGMENT
; Line 2
         PUBLIC   _put
_put     PROC NEAR          「_関数名」でプロシージャ名が定義されている
         push     bp
         mov      bp,sp     関数入口の処理
         xor      ax,ax       ローカル変数の確保(この場合0バイト),残りスタック領域のチェックが,
         call     __chkstk    MS-Cのライブラリルーチン__chkstkで行われる.
; Line 4
;        pos = 4            コメントの形でローカル変数領域における引数の位置が記述されている.
;        col = 6            「;」を削除すれば,MOV AX,[BP+col]のように利用できる*
-------------------------- このあいだにプログラムを挿入すればよい
         mov      sp,bp
         pop      bp        関数を終了する処理
         ret
                                   *「col=6」は「col EQU 6」と同じ効果をもつ.
_put     ENDP
_TEXT    ENDS
END

A>
```

図6-10　CコンパイラでMASMのソースプログラムを出力

ヘッダファイルの利用

本書では，セグメントの設定など，処理系に依存する部分を別ファイルに分離する方法を示しています．プログラムの先頭で，セグメントの設定，定義を行うファイル「PROLOGUE.H」をインクルードし，末尾でセグメントの定義をしめくくるファイル「EPILOGUE.H」をインクルードする，という方法です．これはなるべく処理系に依存しないかたちでプログラムリストを掲載しようという配慮によるもので，読者のみなさんは，前述のコンパイラに雛型(ひながた)を出力させる方法を利用するのが賢明でしょう．

しかし本書の方法は，他のC言語処理系に移植する場合に，これらのインクルードファイルをその処理系用に作りなおすだけで，プログラム本体にはほとんど手を加えずにすむという利点があります．しかも，それらのファイルは前節の方法でコンパイラに出力させたものを一部利用すれば簡単に作成できます．

サンプルプログラム—オセロゲームの思考ルーチン

最後にC言語とリンクするプログラムの具体例として，オセロゲームの思考ルーチンの一部をMASMで記述したものを示します．

オセロゲームなどで有利な手を予測する方法の1つに,先読みがあります．自分が次にここに打ったら相手がたぶんここに打ってきて，という具合にゲームの展開を頭のなかでシミュレートしていくわけです．この方法を，コンピュータならではの高速性を生かして，あらゆる可能性をしらみつぶしに調べ，最も有利な手を選ぶというプログラムです．

とはいっても，1手先，2手先と先に進むごとにゲームの局面の場合の数は幾何級数的に増大します．勝負が決まるまでのすべての展開を予測するためには，とてつもない数の可能性を調べなければならず，実用的ではありません．そこである程度の手数まで先読みをするプログラムを考えます．

有利な手を探すには，プログラムの実行速度が速い方がいいのはいうまでもありません．速度が速ければ，それだけ限られた時間内に多くの場合を調べることができるからです．このように，処理する数が非常に多く，速度を必要とするプログラムは，アセンブリ言語で記述する価値があるといえるでしょう．

チェスやオセロゲームなどの思考プログラムのアルゴリズムはかなり古くから研究されています．ここに示すプログラム例では，基本的なアルゴリズムとされる「ミニマックス法」を利用しています．この方法は，自分が白であるとすれば，自分は常に白の評価値(このプログラムでは白の数)が最大になるような手を打ち，相手は白の評価値が最少になるような手を打ってくるとして先読みを行い，最も有利な手を選ぶ方法です．さらに，探索を効率よく行うために，「$\alpha-\beta$カット」と呼ばれるアルゴリズムも利用しています．探索のアルゴリズムについて解説を始めると，それだけで一冊の本になってしまうので，本書ではこれ以上の解説は行いません．くわしく知りたい方は，巻末に示す参考文献を参照してください．

まず，C言語で記述したプログラムを**リスト6-1**，**リスト6-2**，**リスト6-3**に示します．このうちリスト6-3の部分をMASMで記述しなおしたのが**リ**

スト 6-4 のプログラムです．リスト 6-5 とリスト 6-6 は256ページで解説したインクルードファイルです．なお参考のため，これらのソースプログラムの構成を図 6-11 に示します．

このプログラムのアセンブル，コンパイル，そしてリンク，さらに実行した結果を269ページの図 6-12 に示します．例にしたがってC言語とMASM とのリンクを自分で行ってみてください．

なお，MASM で記述したプログラムは速度よりもわかりやすさを重視しているので，工夫次第でまだまだ速くなります．また，ボードの表示や，人間側の手の入力などのユーザーインターフェイスは，ごく簡単に処理しているので，ぜひ改良に挑戦してみてください．

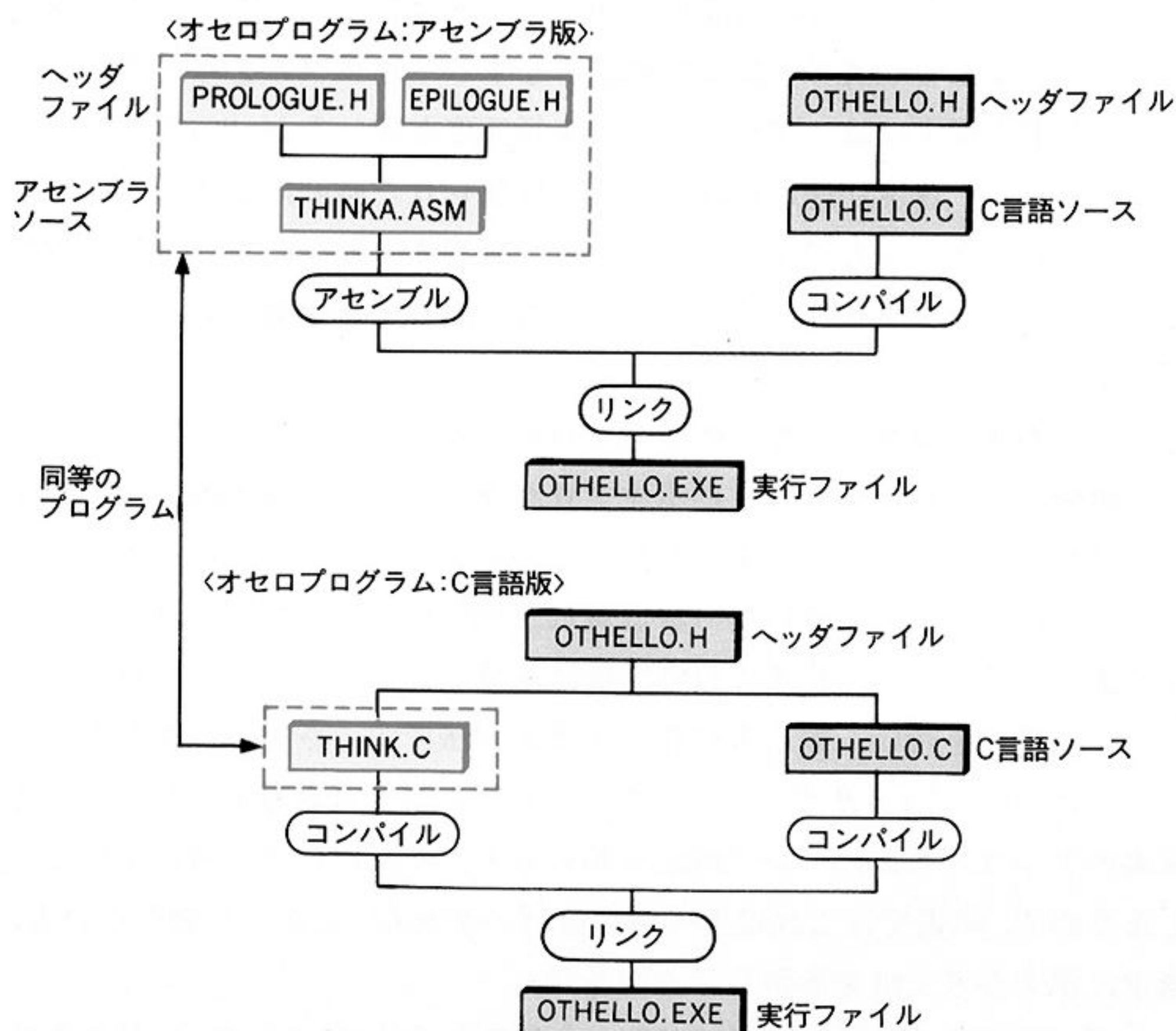



図 6-11　オセロゲームプログラムの構成

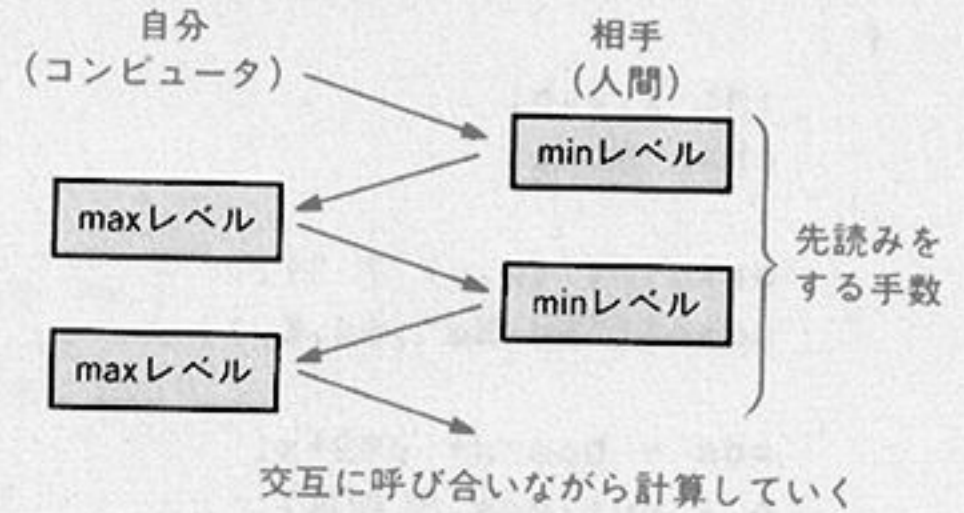
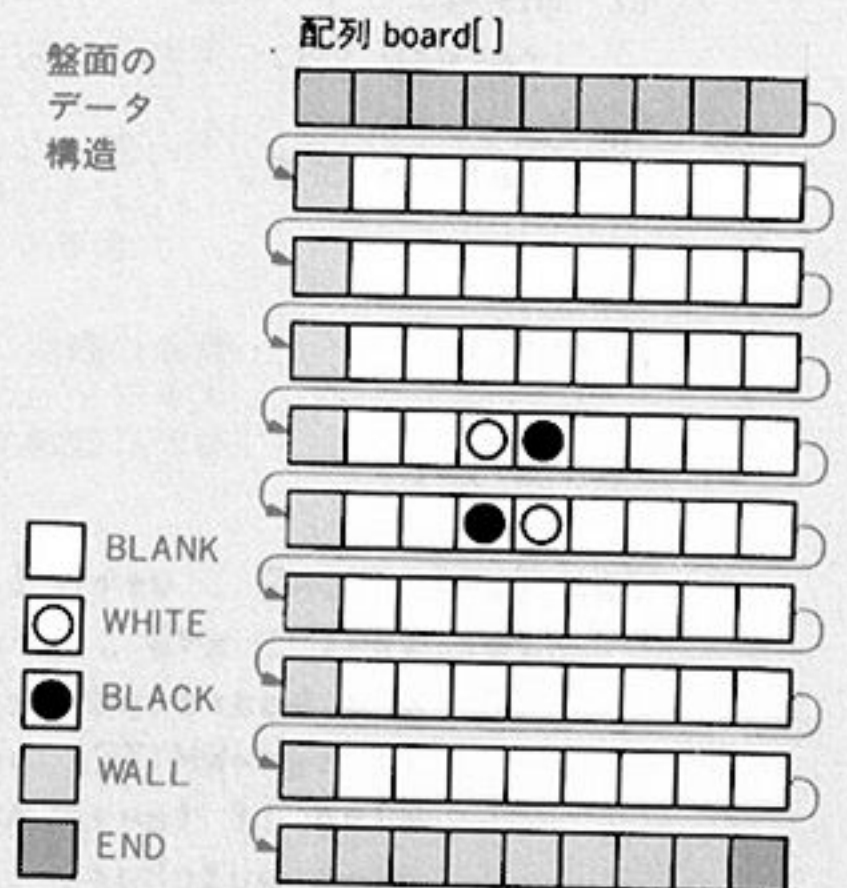
リスト6-1 オセロ・ヘッダファイル OTHELLO.H（C言語）

```
/*
        othello.h
        include file for othello game
*/

#define ON 1
#define OFF 0

#define BLANK 0           ┐
#define BLACK (-1)        │
#define WHITE 1           │ 盤面の状態を表す
#define WALL 2            │
#define END 99            ┘

#define MINVAL (-64)      ┐ 石差の最小値，最大値
#define MAXVAL 64         ┘ この値をともに0にすると，有利な手が1つでも見つかった
                            時点で探索を打ち切る「必勝打ち探索」になる
#define UP_LEFT (-10)     ┐
#define UP (-9)           │
#define UP_RIGHT (-8)     │
#define LEFT (-1)         │ 探索方向
#define RIGHT 1           │ 261ページに示す盤面のデータ構造を参照
#define DOWN_LEFT 8       │
#define DOWN 9            │
#define DOWN_RIGHT 10     ┘

#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))

#define push(r) *sptr++ = (r);  ┐ 探索情報を格納するための仮想的な
#define pop(r) (r) = *--sptr;   ┘ スタック操作の定義
#define MAXSTACK 10000          （ポインタ型とint型の変数を同様に扱うので，
                                  コンパイラによってはwarningを発生する．また
                                  このためスモールモデルでなければならない．）
extern int *sptr; ……………スタックへのポインタ
extern char board[64+9+8+10]; ……………盤面
extern int dif; ……………現在の石数の差を保持する変数
```

リスト6-2 オセロ・メインプログラム OTHELLO.C（C言語）

```
#include <stdio.h>
#include "othello.h"

int *stack,*sptr;                       ┌ level ：先読みの数
int maxlevel;                           │ col   ：石の色
char board[64+9+8+10];           ─────→ │ pass  ：相手が前回，石を置けなかったかどうかを示すフラグ
int dif;                                │ alpha ：自分の最良手で取れる石の数
                                        └ beta  ：相手の最良手で取れる石の数

int max_level(level,col,pass,alpha,beta) ……………マックスレベル（自分の手番）
 int      level,col,pass,alpha,beta;              自分の石数が最高になる位置を計算する
```

```
{
    char    *pos;
    int done;

    if (level==0) return dif; ……………先読みする手数に達したら終了
        done = OFF;

    for (pos=board ; *pos!=END ; pos++) { ……………盤面全体についてチェック
        if (*pos!=BLANK)
            continue;
        if (put(pos,col)) {
            done = ON;
            alpha = max(alpha,min_level(level-1,-col,OFF,alpha,beta));
            putback(col);
            if (alpha>=beta) return alpha; ……………αカット
        }
    }

    if (done!=OFF) return alpha; ………石を置ける位置があった
    if (pass==ON) return dif; ……………石を置ける位置がなく,前回の相手も置けなかったのならば探索終了
    return min_level(level,-col,ON,alpha,beta); ………再度相手の番
}

int min_level(level,col,pass,alpha,beta) ……………ミニレベル(相手の手番)
 int     level,col,pass,alpha,beta;              自分の石数が最小になる位置を計算する
{
    char    *pos;
    int done;

    if (level==0) return dif; ……………先読みする手数に達したら終了
    done = OFF;

    for (pos=board ; *pos!=END ; pos++) { ……………盤面全体についてチェック
        if (*pos!=BLANK)
            continue;
        if (put(pos,col)) {
            done = ON;
            beta = min(beta,max_level(level-1,-col,OFF,alpha,beta));
            putback(col);
            if (beta<=alpha) return beta; ……………βカット
        }
    }
    if (done!=OFF) return beta;
    if (pass==ON) return dif;
    return max_level(level,-col,ON,alpha,beta);
}

char *min_max(col)
 int col;
{
    char *pos,*bestpos;
    int val,maxval;
```

```
    bestpos = NULL;
    maxval = MINVAL;

    for (pos=board ; *pos!=END ; pos++) { ……全ボードの石を置ける位置について,
                                            ひっくり返せる石の数の最大値を求める
        if (*pos!=BLANK)
            continue;
        if (put(pos,col)) { ……相手の手番から始める
            val = min_level(maxlevel,col,OFF,MINVAL,MAXVAL);
            if (val>maxval) {
                maxval=val;
                bestpos = pos;
            }
            putback(col);
        }
    }
    return bestpos;
}

int init() ……初期化ルーチン
{          盤面の初期化, スタックの初期化などを行う
    int i,x,y;

    for (i=0 ; i<91 ; i++)
        board[i] = WALL;

    for (y=1 ; y<9 ; y++)
        for (x=1 ; x<9 ; x++)
            board[y*9+x] = BLANK;
    board[9*9+9] = END;

    board[4*9+4] = WHITE;
    board[5*9+5] = WHITE;
    board[4*9+5] = BLACK;
    board[5*9+4] = BLACK;

    sptr = stack = (int *) malloc(MAXSTACK); ……スタック用の領域を確保
    if (stack==NULL) {
        printf("Can't alloc stack¥n");
        exit(1);
    }

    dif = 0; ……互いの石差, 最初は0
}

int com() ……コンピュータ側の処理, パスならONを返す
{
    char *pos;

    putchar(7); ………
    pos = min_max(WHITE); ……ミニマックス法を実行   ……思考開始と終了のタイミングで,
    putchar(7); ………                                  ベルを鳴らしている
    if (pos==NULL)
```

自分（コンピュータ）
相手（人間）
minレベル
maxレベル
minレベル
maxレベル
先読みをする手数
交互に呼び合いながら計算していく

盤面のデータ構造
配列 board[ ]
BLANK
WHITE
BLACK
WALL
END

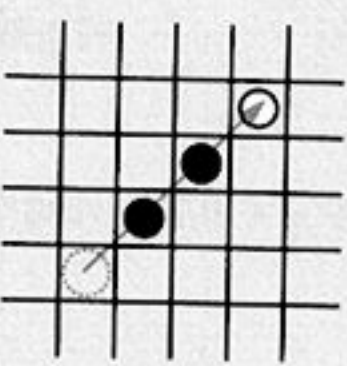
```
        return ON;
    put(pos,WHITE); ………石を置く
    sptr = stack; ……………スタックにひっくり返した石の位置が
    return OFF;              記録されているのでクリアする
}


int man() ……………人間側の処理．パスならONを返す
{
    int x,y,n;
    char *pos;

    printf("(x,y) ? ");      } キーボードから位置を指定する
    scanf("%d %d",&x,&y);    } (カーソルやマウスで位置を指定
                               できるようにするとよい)
    pos = board+ y*9+x;
    n = put(pos,BLACK); ……………石を置く
    sptr = stack;
    if (n==0)
        return ON;      相手の石を返せない位置を指定することで，
    else                パスを入力する
        return OFF;     (置けない位置を入力すると，再度入力させる
}                        ようにした方がよい．パスの判定は自動化で
                         きるので，ぜひ改良してほしい)
int print() ……………盤面の表示
{                 (グラフィックを利用した表示
                   などに改良するとよい)
    int x,y,s;

    for (y=1 ; y<9 ; y++) {
        for (x=1 ; x<9 ; x++) {
            s = board[y*9+x];
            if (s==WHITE) putchar('X');
            else if (s==BLACK) putchar('O');
            else putchar('.');
        }
        putchar('¥n');
    }
    putchar('¥n');
}

void main()
{
    int pass;

    init();

    printf("level ? ");          } 先読みする手数を入力
    scanf("%d",&maxlevel);       }

    print();
```

```
    pass = Ø;
    for (;;) {
        if (man()==ON) {
            printf("Pass...¥n");
            if (++pass>=2)
                break;
        } else {
            print();
            pass = Ø;
        }
        if (com()==ON) {
            printf("Pass...¥n");
            if (++pass>=2)
                break;
        } else {
            print();
            pass = Ø;
        }
    }
}
```

人間側の処理

コンピュータ側の処理

リスト 6-3 オセロ・思考サブルーチン THINK.C (C言語)

```
/*
    think.c
*/
#include "othello.h"

int search(pos,dir,col)
 char *pos;
 int dir,col;
{
    int n,i,ocol;

    ocol = -col;
    n = Ø;
    while( *(pos+=dir)==ocol )
        n++;

    if (n==Ø || *pos!=col)
        return Ø;
    for (i=n ; i>Ø ; i--) {
        *(pos-=dir) = col;
        push(pos);
    }
    return n;
}

int put(pos,col)
 char *pos;
 int  col;
```

int search(pos,dir,col) …… 自分の石を置きたい位置と，探索方向，石の色を指定して相手の石をひっくり返せる数を返す

while(*(pos+=dir)==ocol) …… 相手の石以外が，見つかるまでその方向をチェック

if (n==Ø || *pos!=col) …… はさめる相手の石がなかったり，その反対側に自分の石がなかったら，ひっくり返せない

for (i=n ; i>Ø ; i--) { …… 相手の石をひっくり返し，その位置をスタックに記録する

int put(pos,col) …… 自分の石を置きたい位置と石の色から，相手の石をひっくり返せる数を返す

```
{
    int n;

    n  = search(pos,UP_LEFT,col);
    n += search(pos,UP,col);
    n += search(pos,UP_RIGHT,col);
    n += search(pos,LEFT,col);         各方向へ探索し,
    n += search(pos,RIGHT,col);        ひっくり返せる数の合計を計算する
    n += search(pos,DOWN_LEFT,col);
    n += search(pos,DOWN,col);
    n += search(pos,DOWN_RIGHT,col);

    if (n==0)
        return 0;    合計値が0なら,その位置には置けない

    *pos = col;……………………その位置に自分の石を置く
    dif += (n*2+1)*col;…………現在の石数の差を計算する
    push(pos);…………………………自分の石を置いたことをスタックに記録する
    push(n);……………………………置いた石の数をスタックに記録する

    return n;
}
int putback(col)…………探索のためにひっくり返した石をもとに戻す
 int col;
{
    char *pos;
    int n,ocol;

    pop(n);
    pop(pos);…………置いた石の数をスタックから取り出す
    ocol = -col;………自分の石を置いた位置をスタックから取り出す

    dif -= (n*2+1)*col;…………現在の石数の差をもとに戻す
    *pos = BLANK;…………………石を取り除く

    while (n--) {
        pop(pos);       ひっくり返した相手の石をもとに戻す
        *pos = ocol;
    }
}
```

リスト 6-4 オセロ・思考サブルーチン THINKA.ASM (MASM)

C言語とのリンクに関する部分のみ解説を示す. プログラムのアルゴリズムに関しては, リスト6-3(THINK.C)を参照

```
        EXTRN    _dif:WORD,_sptr:WORD
 INCLUDE PROLOGUE.H

ARG1      EQU     4
ARG2      EQU     6
ARG3      EQU     8

ON        EQU     1
OFF       EQU     Ø

BLANK     EQU     Ø
BLACK     EQU     (-1)
WHITE     EQU     1
WALL      EQU     2
LAST      EQU     99

MINVAL    EQU     (-64)
MAXVAL    EQU     64

UP_LEFT EQU       (-1Ø)
UP        EQU     (-9)
UP_RIGHT EQU      (-8)
LEFT      EQU     (-1)
RIGHT     EQU     1
DOWN_LEFT EQU     8
DOWN      EQU     9
DOWN_RIGHT EQU    1Ø

_search PROC      NEAR
;         SI : pos
;         DX : dir
;         AH : col
;         DI : STACK PTR
          PUSH    SI
          PUSH    BX
          PUSH    CX
          XOR     BX,BX
S_WHILE:
          ADD     SI,DX
          MOV     AL,[SI]
          ADD     AL,AH
          JNZ     S_BREAK
          INC     BL
          JMP     SHORT S_WHILE
S_BREAK:
          MOV     CX,BX
```

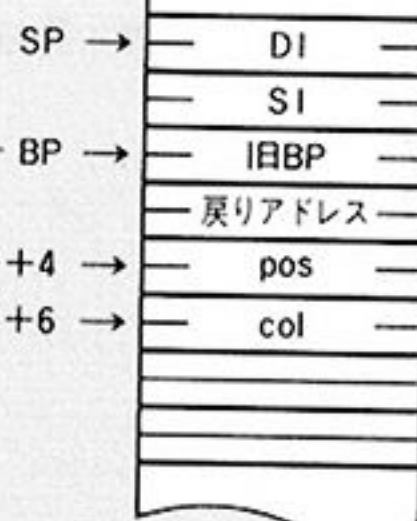
EXTRN: C言語で関数の外に定義されている大域変数を参照するためにEXTRN宣言する. セグメントの外で定義しなければならないことに注意(225ページ参照)

ARG1〜ARG3: C言語からの引数を受けとるために, スタック領域でBPレジスタの指すアドレスからの距離を定義しておく

LAST: MASMではENDは擬似命令なので使えない. そこでLASTという名前に変更した

ON〜DOWN_RIGHT: C言語側と同様の定数定義

_search: このルーチンはモジュール内でしか使われないので, パラメータをレジスタで受け渡しする

```
        JCXZ    S_RETURN
        CMP     [SI],AH
        JE      S_TURN
        MOV     BL,Ø
        JMP     SHORT S_RETURN
S_TURN:
        SUB     SI,DX
        MOV     [SI],AH
        MOV     [DI],SI
        INC     DI
        INC     DI
        LOOP    S_TURN
S_RETURN:
        MOV     AL,BL
        POP     CX
        POP     BX
        POP     SI
        RET
_search ENDP
-------------------------------------------------------------
_put    PROC    NEAR
        PUBLIC  _put
; int     put(pos,col)
;  char    *pos;
;  int     col
;
        PUSH    BP
        MOV     BP,SP
        PUSH    SI
        PUSH    DI
        ;
        MOV     SI,ARG1[BP]
        MOV     AH,ARG2[BP]
        MOV     DI,_sptr
        XOR     CX,CX
        ;
        MOV     DX,UP_LEFT
        CALL    _search
        MOV     CL,AL
        MOV     DX,UP
        CALL    _search
        ADD     CL,AL
        MOV     DX,UP_RIGHT
        CALL    _search
        ADD     CL,AL
        MOV     DX,LEFT
        CALL    _search
        ADD     CL,AL
        MOV     DX,RIGHT
        CALL    _search
        ADD     CL,AL
        MOV     DX,DOWN_LEFT
        CALL    _search
```

```
          ADD     CL,AL
          MOV     DX,DOWN
          CALL    _search
          ADD     CL,AL
          MOV     DX,DOWN_RIGHT
          CALL    _search
          ADD     CL,AL
          ;
          JCXZ    P_RETURN
          ;
          MOV     [SI],AH
          MOV     BX,CX
          SHL     BX,1
          INC     BX
          CMP     AH,0
          JGE     P_WHITE
          NEG     BX
P_WHITE:
          ADD     _dif,BX      ……C言語で定義した大域変数difを参照する.
                                 名前の先頭に「_」が付くことに注意
          MOV     [DI],SI
          INC     DI
          INC     DI
          MOV     [DI],CX
          INC     DI
          INC     DI
P_RETURN:
          MOV     _sptr,DI     ……C言語で定義した大域変数
          MOV     AX,CX
          POP     DI
          POP     SI
          POP     BP
          RET
_put      ENDP
------------------------------------------------------------------------
_putback            PROC       ←――C言語から呼び出せるプロシージャ_putback
          PUBLIC  _putback
; void    putback(col)
;   int    col
;
          PUSH    BP           ……引数を参照するための処理
          MOV     BP,SP          BPレジスタの内容は保存しなければならない
          PUSH    SI           ……MS-CではSIおよびDIレジスタをレジスタ変数
          PUSH    DI             として使用するので,その値を保存しなければ
          ;                      ならない
          MOV     AH,ARG1[BP]
          MOV     DI,_sptr
          DEC     DI
          DEC     DI
          MOV     CX,[DI]
          DEC     DI
          DEC     DI
          MOV     SI,[DI]
```

```
        ;
        MOV     BX,CX
        SHL     BX,1
        INC     BX
        CMP     AH,Ø
        JGE     B_WHITE
        NEG     BX
B_WHITE:
        SUB     _dif,BX
        MOV     [SI],BYTE PTR BLANK
        ;
        NEG     AH
B_WHILE:
        DEC     DI
        DEC     DI
        MOV     SI,[DI]
        MOV     [SI],AH
        LOOP    B_WHILE
        ;
        MOV     _sptr,DI
        POP     DI
        POP     SI
        POP     BP
        RET
_putback        ENDP
-----------------------------------------
INCLUDE EPILOGUE.H
        END
```


リスト 6-5　セグメント定義用インクルードファイル PROLOGUE.H（MS-C 用）

```
_TEXT   SEGMENT  BYTE PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
DGROUP  GROUP   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
_TEXT     SEGMENT
```

C 言語の生成するセグメントの定義等を抜き出したもの

中身のないセグメント定義があるが，これは_TEXT，_DATAという名前がセグメント名であることを宣言する，セグメントをこの順で配置するという役割がある

コードセグメントの定義

こうしておけば，プログラム本体ではセグメント定義に関して何も注意を払わなくてよい

リスト 6-6　セグメント定義用インクルードファイル EPILOGUE.H（MS-C 用）

```
_TEXT   ends
```

セグメント定義の終わり

```
            ┌──ラベル名，シンボル名の大文字と小文字を区別する
A>MASM /ML THINKA ;↵ ……………Cのプログラムとリンクする場合には，MLオプションを付けて
Microsoft MACRO Assembler  Version 3.00              アセンブルすること
(C)Copyright Microsoft Corp 1981, 1983, 1984

49118 Bytes free

Warning Severe
Errors  Errors
0       0

A>CL OTHELLO.C THINKA↵ ……………Cプログラムのコンパイルとリンクを行う(MS-Cの場合)
Microsoft (R) C Compiler  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985, 1986.  All rights reserved.

OTHELLO.C
Microsoft (R) Overlay Linker  Version 3.51
Copyright (C) Microsoft Corp 1983, 1984, 1985, 1986.  All rights reserved.

Object Modules [.OBJ]: OTHELLO.OBJ THINKA
Run File [OTHELLO.EXE]: OTHELLO.EXE/NOI      …MS-CのCLコマンドは，/NOIオプ
List File [NUL.MAP]: NUL                      ションを付けてLINKコマンドを呼
Libraries [.LIB]: ;                           び出す．/NOIオプションにより，
                                              LINKはパブリックなラベルの大文
                                              字・小文字を区別する．
A>OTHELLO↵ ……………実行してみる
level ? 3↵ ……………先読みする手数を入力する
........
........
........
...XO...
...OX...
........
........
........

(x,y) ? 4 3↵ ……………人間の手番
........
........
...O....
...OO...
...OX...
........
........
........

........ ……………コンピュータの手番
........
..XO....
...XO...
```

図6-12 オセロゲームの作成手順と実行例

COLUMN

C言語とアセンブラの統合

C言語のプログラムとアセンブラのプログラムをリンクする方法は，本文で解説したように，それほど複雑ではありませんが，いろいろなところに気を使わなければならず面倒です．このため，アセンブラのソースプログラムを書きやすくするためのプログラミング言語や各種のユーティリティが発売されています．ここではC-MASM(㈱ライフボート)とTurbo-C(㈱マイクロソフトウェア アソシエイツ，㈱サザンパシフィック)を紹介しましょう．

C-MASMは，いくつかのヘッダファイルとして提供されます．そのファイルには5章で解説したマクロ命令を使って，便利な命令が数多く定義されています．そのヘッダファイルを自分のプログラムにインクルードすれば，下の図のようにC言語とのリンクをサポートする各種の命令を利用できます．

```
INCLUDE EZ_ASM.MAC ……………C-MASMのヘッダファイルをインクルードする

DEFINE_MODULE   thinka ……………モジュールの定義
                        セグメントの定義や指定は必要ない

        EXTERN DATA_PTR , sptr  ┐外部参照の変数を宣言   C言語での名前で定義できる
        EXTERN _INT , dif       ┘                        (_を付けなくてよい)
        START_FCN put < <DATA_PTR,pos> , <_INT,col> > ……関数の定義
        PROLOGUE <SI,DI> ………レジスタの保存

        mov     si,pos  ┐関数の引数の型と名前を宣言しておくと，
        mov     ah,col  ┘その名前でアクセスすることができる．
        mov     di,sptr   また，スタックフレームの構造を考えなくてよい
        xor     cx,cx

        mov     [si],ah
        mov     bx,cx
        shl     bx,1
        inc     bx
        .if  <ah> GE 0 , then_goto p_white ……………C言語に似た制御構造を
        neg     bx                                  用いることができる
p_white:        └──MASMでは，シンボルの先頭に「.」(ピリオド)を付けることも可能
```

Turbo-Cは，C言語処理系をトータルなプログラミング環境として提供するという点で最も注目されているCコンパイラの1つです．その豊富な機能のなかに，C言語のプログラムからCPUのレジスタを操作したり，アセンブリ言語のニーモニックを記述できるという機能も用意されています．これはかなり強力な機能で，アセンブラのプログラムを完全にCコンパイラの管理下で記述できるので，ごく自然にC言語とアセンブラのリンクを実現できます．そのため，下の図に示すようにC言語で定義した変数をそのままアセンブラのプログラムでアクセスすることも可能となります．

```
extern int dif,sptr;


#pragma inline ……………インラインアセンブラ機能を有効にする

put(pos,col)
 char *pos;
 int  col;
{
        asm mov si,pos; ｝「asm␣ニーモニック；」の形でCのソースプログラム
        asm mov ah,col; ｝中にマシン語命令を記述することができる
        _DI = sptr;     ｝8086CPUのレジスタを変数としてアクセスすること
        _CX = 0;        ｝ができる

        if ((char) _AH < 0 ) ……………レジスタはunsigned int／char型
               asm neg bx;              として扱われる
        dif = _BX;
}
```

注：インラインアセンブラ機能を利用できるのはTurbo-Cのコマンドライン版(tccコマンド)のみ
　　また，MASM Ver.4.0以降が必要．

ハードウェア割り込み

2.3章で解説したように，割り込み処理は高級言語で扱うことは難しく，ほとんどの場合アセンブリ言語で記述されます．そこでMASMのプログラミング例として，割り込みを利用したプログラムを紹介しましょう．

ここで扱う割り込みとは「ハードウェア割り込み」のことで，CPUの動作とは非同期に発生する入出力要求を処理するための機能です．キーボード入力を例にとって，割り込みの仕組みを図6-13に解説しました．くわしい解説

キー入力
①
8251 キーボードインターフェイス
②
8259 割り込みコントローラ
③タイプ09H
<CPU>
IP:
CS:
④CS,IP, およびフラグレジスタの内容をスタックへ退避
⑤割り込みベクタテーブルからタイプ09Hの割り込みベクタをロード
⑥CS,IPレジスタにより示される割り込み処理ルーチンを実行
⑦割り込みルーチンが終了すると，スタックからCS, IP, およびフラグレジスタを復帰し，もとの作業に戻る
<メモリ>
物理アドレス
00000H オフセットアドレス
00002H セグメントアドレス
00004H オフセットアドレス
00006H セグメントアドレス
00024H オフセットアドレス
00026H セグメントアドレス
003FCH オフセットアドレス
003FEH セグメントアドレス
XXXXXH タイプ09Hの割り込み処理ルーチン
ESC.COM
タイプ0Hの割り込みベクタ
タイプ1Hの割り込みベクタ
タイプ09Hの割り込みベクタ
タイプFFHの割り込みベクタ
割り込みベクタテーブル
メモリ上に固定的に1Kバイト確保される
MS-DOSの常駐部
ユーザープログラム
スタックエリア

図6-13　ハードウェア割り込みの仕組み

は「はじめて読む8086」など，他書を参考にしてください．

ハードウェア割り込みの発生により呼び出されるプログラム，すなわち割り込み処理ルーチンを作成するために必要な知識や注意点を以下に示します．

割り込みプログラムの登録

割り込みが発生したときに，それを処理するプログラムが実行されるようにするには，割り込み処理ルーチンのアドレス，すなわち割り込みベクタを割り込みベクタテーブルに登録します．割り込みベクタテーブルはセグメントアドレス 0000_H のオフセットアドレス 0000_H から $03FF_H$ までの1Kバイトの領域です．4バイトごとに割り込み処理ルーチンのアドレスが格納され，それぞれオフセットアドレス，セグメントアドレスの順に並んでいます．つまり，割り込み処理ルーチンのアドレスは，割り込み番号を4倍したオフセットアドレスに格納すればよいことがわかります(前述の図6-13を参照)．

2章の57ページのコラムで解説したように，セグメントアドレスとオフセットアドレスを同時に格納することはできませんから，この間割り込みを禁止しなければなりません．どちらか一方だけ格納した状態で割り込みがかかると，不用意なところへ飛んでいってしまうからです．

ハードウェア割り込みの処理は，一見簡単ですが，1つ間違えると危険な結果になりかねないので十分な注意が必要です．このため，割り込みベクタの登録機能がMS-DOSのファンクションコールにも用意されています．それを利用するのが最も簡単で確実でしょう．本書でもこの方法を利用しています*．

システムの初期化

割り込み処理ルーチンの登録をすませたら，次に割り込みに関係する装置を初期化して，割り込みが正しく発生するようにしなければなりません．初

*富士通FM-16β，FMRシリーズは，ROM BIOSに割り込みを管理する機能が用意されているので，これを利用するとよい．

期化の対象となるのは，「割り込みコントローラ」と割り込みを発生する「入出力インターフェイス」です．

割り込みコントローラ全体の初期化は，通常，システム起動時に行われておりあらためて設定する必要はありません．必要なのは，「割り込みマスクレ

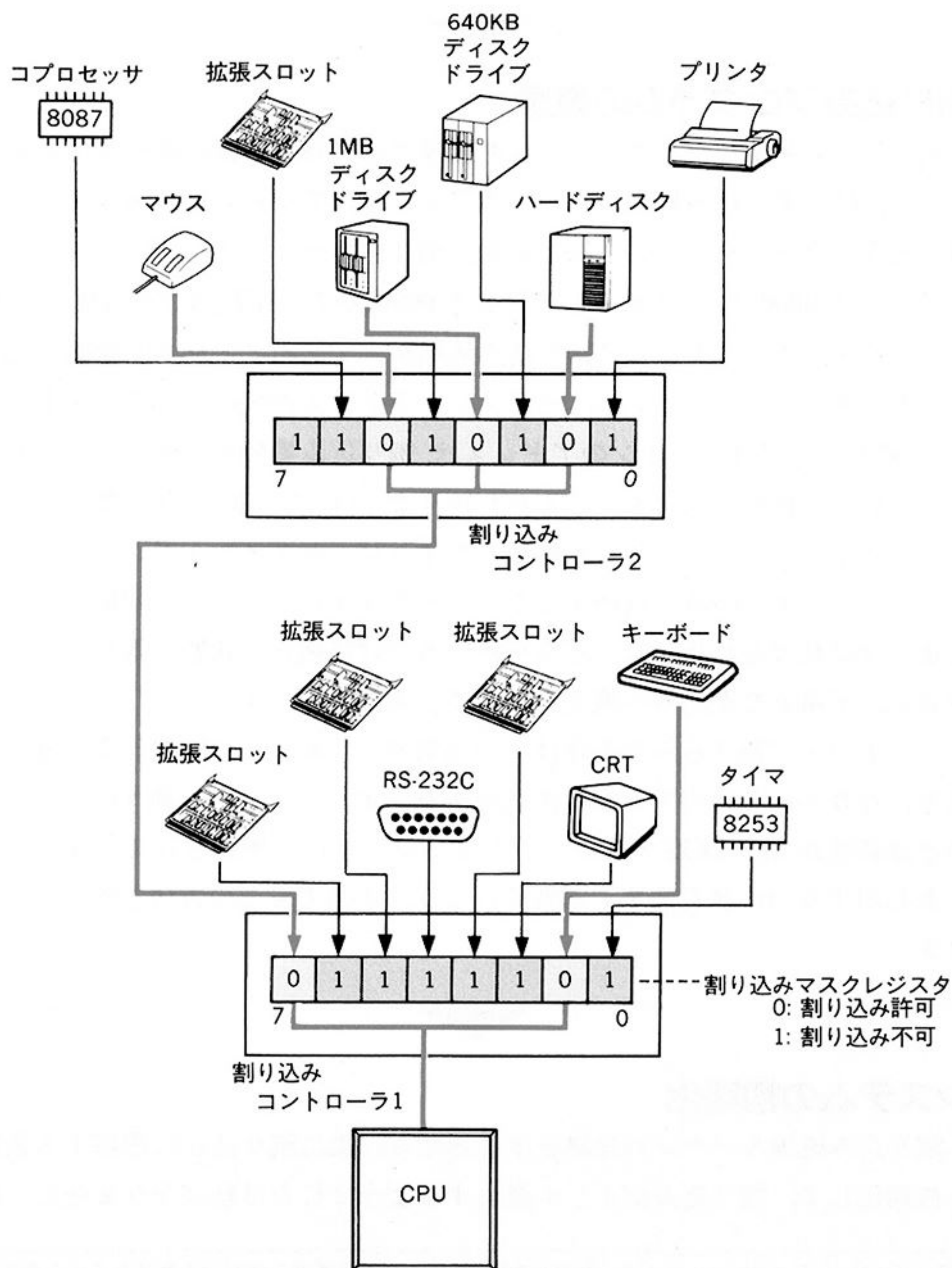



図 6-14　割り込みコントローラの役割

ジスタ」の設定だけです．

割り込みマスクレジスタは，割り込みコントローラのなかにあるレジスタです．このレジスタは，図 6-14 に示すように割り込み信号を通過させるかどうかを制御する関所の働きをします．割り込みマスクレジスタの各ビットは，割り込みを発生させる装置に割り当てられています．ビットが「1」なら割り込みはマスクされ，CPU には伝えられません．割り込み信号を CPU まで伝えるためには，各装置に対応するビットを「0」にしなければなりません．

参考までに NEC PC-9801 シリーズについて，割り込みマスクレジスタの各ビットと装置との対応を図 6-14 に示します．

次に割り込みをかける装置自身を初期化します．各種入出力インターフェイスはプログラマブルになっていることが多く，入出力の要求を CPU に伝えるために割り込みをかけるかどうかを設定できるようになっています．そこで割り込みがかかるようにインターフェイスの設定を行います．

この設定は機器によって異なり，なかには複雑なものもあります．くわしくは各機種のハードウェアマニュアルや解説書を参考にしてください．

割り込み処理ルーチン

割り込み処理ルーチンは一種のサブルーチンですが，いくつかの点に注意して記述しなければなりません．

レジスタの保存

割り込み処理ルーチンは，他のプログラムを実行中に突然呼び出されます．もとのプログラムに戻ったときにそのまま続きを実行できるようにするためには，すべてのレジスタを呼び出された時点と同じ状態にしてから戻らなければなりません．このため割り込み処理ルーチンの入り口では，使用するレジスタをすべてスタックなどに保存します．そして割り込み処理ルーチンを抜ける前にそこから取り出してもとの値に戻すのです．

IRET 命令

割り込み処理ルーチンは一種のサブルーチンであると述べましたが，リ

ターン命令は通常のサブルーチンにおけるRET命令ではなく，IRET命令を用います．IRET命令の動作については2章57ページのコラムで解説していますので，もう一度読み返してみるとよいでしょう．

スタックの使用

レジスタの保存などにはスタックを使用するのが最も便利なのですが，あまり使いすぎると危険です．割り込み処理ルーチンはどこで呼び出されるかわからず，呼び出された時点で動いているプログラムにスタック領域の余裕があるかどうかわからないからです．サブルーチン呼び出しなどでスタックを使いすぎると，PUSHしたデータがスタック領域を溢れ出してデータ領域などを破壊してしまうかもしれません．

スタックの容量に不安がある場合は，割り込み処理ルーチンのなかで独自にスタック領域を確保し，切り替えて使うことになります．

MS-DOS，ROM BIOSの利用

一般に，割り込み処理ルーチンからはMS-DOSやROM BIOSを呼び出すことはできません．なぜなら，MS-DOSやROM BIOS中のルーチンを実行中に割り込みが発生する可能性があるからです．割り込み処理ルーチンからそれらのルーチンを呼び出すと，内部の変数などを使用して内容が変化してしまい，割り込み処理ルーチンを終了してもとのルーチンに戻るとたいへん危険です．割り込み処理ルーチンでは，割り込みが発生した時点で実行中であったプログラムの状態を変化させてはいけません．

なお，ROM BIOSは多くのサブルーチンの集まりですが，そのなかには内部に変数領域を持たないものもあります．そのことを確認すれば，割り込み処理ルーチンからそのルーチンを呼び出すことができます．本書のプログラムでも，割り込み処理ルーチンから呼び出して安全なものを利用しています．

EOIの発行

割り込み処理ルーチンを終了する前に，ハードウェア的に割り込み状態を終了させる処理をしなければなりません．それは割り込みコントローラに割り込みが終了したことを伝える処理です．この処理は「EOI(End Of Inter-

rupt)の発行」と呼ばれています．

割り込みコントローラは，割り込み処理ルーチン実行中はそれより優先度の低い割り込みを受け付けないなど，各種の割り込みに関する制御を行います*．他の割り込みを再び受け付ける状態にするには，割り込み処理ルーチンが終了したことを割り込みコントローラに伝えなければなりません．そのために行うのがEOIの発行です．

サンプルプログラム—スクリーンセーバー

割り込みを使ったプログラムの例としてスクリーンセーバーを取り上げます．これはCRT画面を保護するためのプログラムです．CRT画面はその性質上，同じ映像を長時間表示しつづけると明るい部分が焼きついてしまいます．それを防止するために一定時間画面表示が変化しなければ，自動的に画面を暗くしてしまうというプログラムがスクリーンセーバーです．ちょっと席をはずすときなど，パソコンをそのままにしておいても安心できるオシャレなツールです．

画面表示を監視するのは難しいので，代わりにキーボードを監視し，一定時間キー入力がなければ画面表示も変化していないと判断します．そして，単に画面を暗くするだけでは，誰か通りかかった人に使用されていないと思われてパソコンの電源を切られたりする恐れがあるので，ちょっとしたアニメーションを表示させます．映像が動いていれば焼きつく心配はありません．

使用する割り込みは一定時間ごとに発生するタイマー割り込みと，キーボード割り込みです．プログラム例はNEC PC-9801シリーズ用に記述したものなので，他機種を使用している方はマニュアル等を参考にして機種に依存する部分を改造してください．なお，FMR-60/70シリーズ用については差分リストを掲載します．

PC-9801ではタイマー割り込みはPRINTコマンドなどに使用されるので，代わりにVSYNC割り込みを利用します．これはCRTコントローラが

＊274ページの図6-14でいえば，右側の機器ほど割り込み優先度が高い．優先度の高い割り込みに時間がかかると，他の割り込みの処理に支障が生じる．たとえば，RS-232C回線から送られたデータの取りこぼしなどが起こる．

画面リフレッシュを開始するタイミングで1秒間に60回発生させる割り込みなので，タイマー割り込みと同じように利用することができます．

もう1つはキー割り込みですが，これを勝手に利用すると通常のキー入力ができなくなってしまいます．そこで，割り込み処理ルーチンの入り口だけを乗っとるというテクニックを使います*．くわしくは図6-15を参照してください．この方法はいろいろな応用が考えられるでしょう．

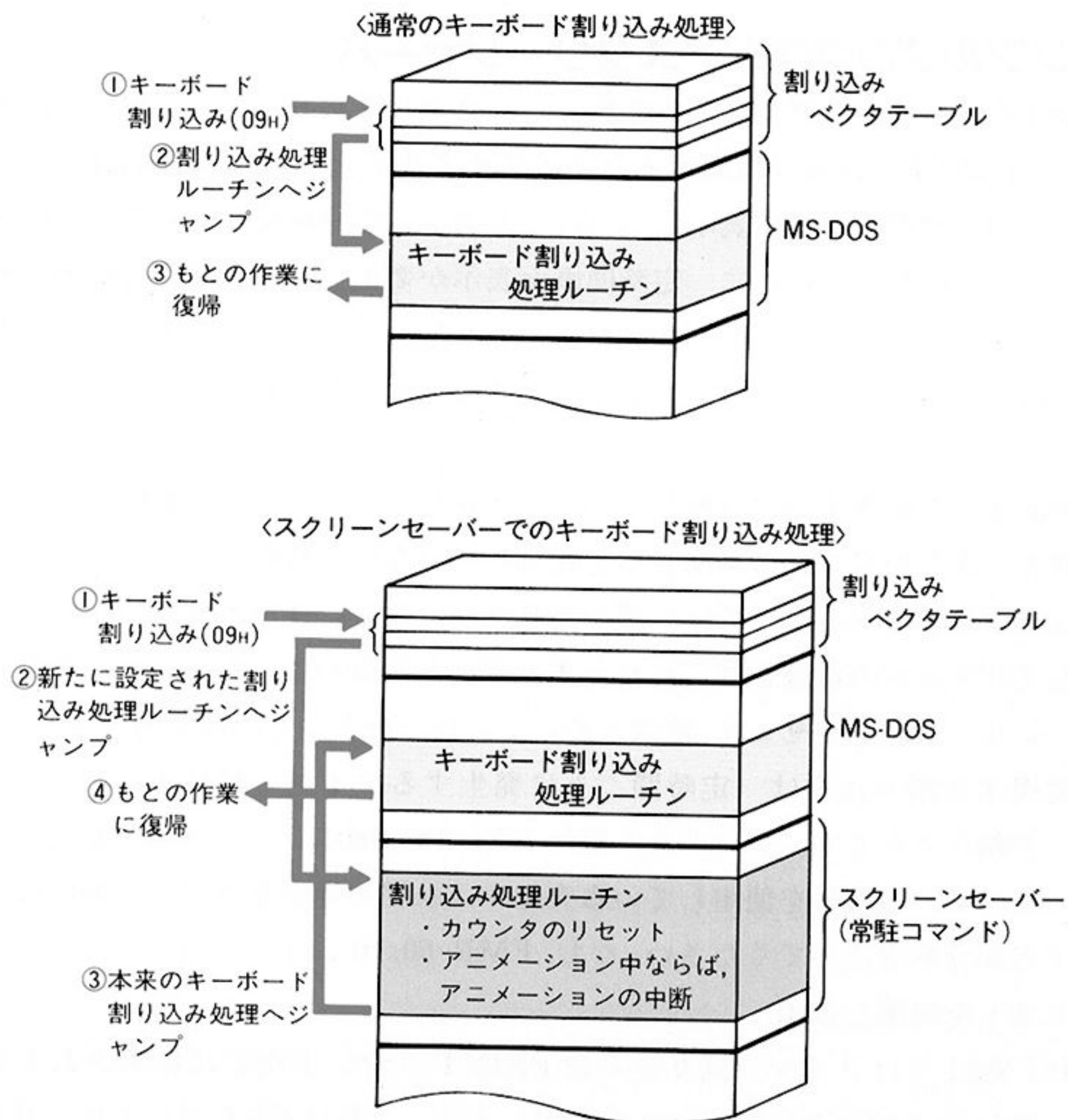



図6-15　割り込み処理ルーチンの乗っとり

*この方法を割り込み処理ルーチンに「フックをかける」という．

VSYNC割り込み処理ルーチンでは，割り込みごとにカウンタをデクリメントします．キー割り込み処理ルーチンではそのカウンタを初期値に戻します．したがって，カウンタが0になれば一定時間キー入力がなかったと判断することができます．

その時点で画面表示を停止し，アニメーション表示を開始します．アニメーション画面は引き続きVSYNC割り込みが発生するごとに更新し，キー割り込みが発生した時点で中止して，もとの画面表示を再開します．

このプログラムではMS-DOSの機能の1つである，常駐終了のテクニックを利用しています．割り込み処理ルーチンの登録など，初期化の処理が終わっても割り込み処理ルーチンはそのままメモリにとどまらなければなりません．このための機能が常駐終了です．スクリーンセーバーを起動すると，すぐに何事もなかったかのように終了してしまいますが，割り込み処理ルーチンは動き続けています．そして，他のプログラムを実行中でもカウントを続けて，役目を果たします*．

なお，画面表示を停止するのはテキスト画面だけで，グラフィック画面には何も表示されていないことを前提としています．そして，アニメーション表示はグラフィック画面に対して行います．グラフィック処理の例としても参考になると思いますので，ぜひ解読に挑戦してください．

プログラムの分量の関係から，アニメーションにはごく簡単なものを掲載していますが，いろいろ工夫のしがいがあるところです．おもしろいアイディアを考案して，改良してみてください．過去には，打ち上げ花火のアニメーションや，ライフゲーム，虫が這い回るものなどがありました．

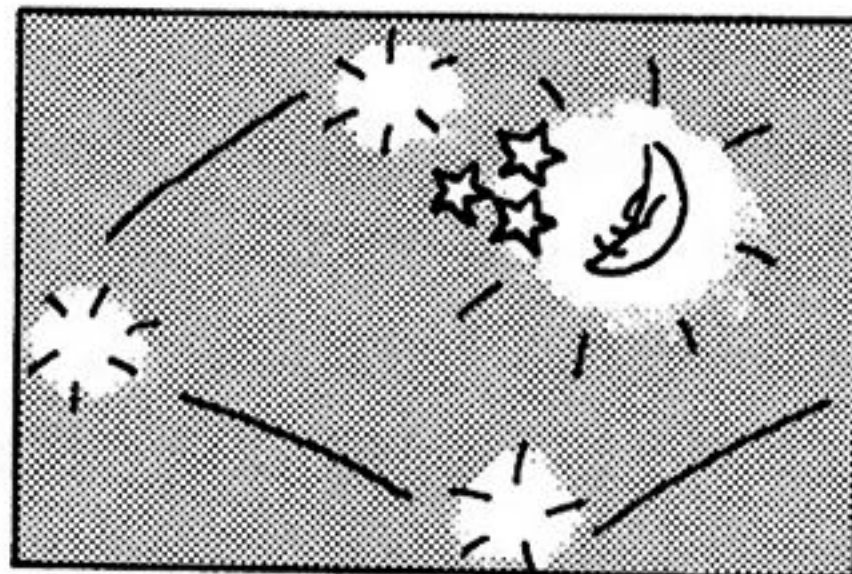
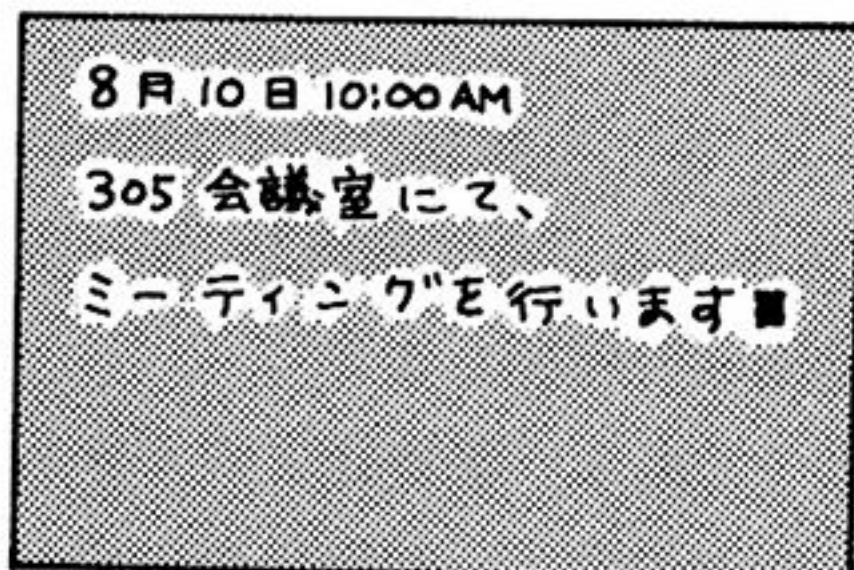
このプログラムのNEC PC-9801シリーズ用のソースリストを**リスト6-7**に示します．また，富士通FMR-60/70用の差分リストを**リスト6-8**に掲載します．プログラムはCOMモデルとして作成しましたので，アセンブル&リンクの方法は1章で解説したとおりです．各自でやってみてください．

*アプリケーションプログラムによっては，VSYNC割り込みを初期化してしまうものがある．このようなプログラム上では本書のスクリーンセーバーは動作しない．

しばらくキーを押さないと，
スクリーンセーバーが起動する．

→

8月10日 10:00 AM
305 会議室にて、
ミーティングを行います

←

何かキーを押すと，
もとの画面に戻る．

リスト6-7　スクリーンセーバー SS.ASM

```
 1: ;           SS.ASM
 2: ;           SCREEN SAVER VER 1.0      BY T.KAMACHI
 3: ;
 4:
 5: TIMER EQU          10 ............スクリーンセーバーが働き出すまでの時間(秒単位)
 6:                                  この値を変更することにより時間を調整できる(最大値1092秒)
 7: RTIMER   EQU       60*TIMER      ここでは結果を確認しやすいように短かくしている
 8: MIN_X    EQU       0      ┐
 9: MAX_X    EQU       640    │ キャラクタの動く範囲
10: MIN_Y    EQU       0      │ CRTC割り込み時に表示しているので，画面リフレッシュと
11: MAX_Y    EQU       400    ┘ 同期して画面上部ではちらついてしまう
                                MIN_Yを調整すればこれを防げる(PC-9801版)
12:
13: CODE     SEGMENT
14:          ASSUME CS:CODE,DS:CODE
15:          ORG 100H
16: START:
17:          JMP       INIT ............初期化ルーチンはプログラム末尾にある
                                       (常駐終了時に切り離すため)
18: ;   ワークエリア
19: KEYVEC   DD        ? ...............キーボード割り込みルーチンのアドレス保存領域
20: COUNT    DW        RTIMER .........キーボードが押されないあいだ，カウントダウンされる
21: X        DW        MAX_X/2  ┐
22: Y        DW        MAX_Y/2  ┘ キャラクタ表示位置
23: DELTAX   DW        1  ┐
24: DELTAY   DW        1  ┘ キャラクタ移動量
25:
26: ;        GRAPHIC DATA PATTERN ............キャラクタのグラフィックデータの定義
                                                     横            縦
27: WIDTH_X EQU        2   ┐
28: WIDTH_Y EQU        16  ┘ キャラクタの大きさ  2バイト(16ドット)×16ドット
```

```
29: PATTERN_BUF      DW      WIDTH_Y,WIDTH_X
30:            ;   BLUE PLANE DATA
31:            DB      00000000B,00000000B
32:            DB      00000000B,00000000B
33:            DB      00000000B,00000000B
34:            DB      00000000B,00000000B
35:            DB      00000000B,00000000B
36:            DB      00000000B,00001000B
37:            DB      00000000B,00010100B
38:            DB      00000000B,00001000B
39:            DB      00000000B,00000000B
40:            DB      00000000B,00000000B
41:            DB      00000000B,00000000B
42:            DB      00000000B,00000000B
43:            DB      00000000B,00000000B
44:            DB      00000000B,00000000B
45:            DB      00000000B,00000000B
46:            DB      00000000B,00000000B
47:            ;   RED PLANE DATA
48:            DB      00000000B,11000000B
49:            DB      00000000B,01110000B
50:            DB      00000000B,00111100B
51:            DB      00000000B,00011110B
52:            DB      00000000B,00011110B
53:            DB      00000000B,00011110B
54:            DB      00000000B,00010111B
55:            DB      00000000B,00011111B
56:            DB      00000000B,00111111B
57:            DB      00000000B,00011011B
58:            DB      01000000B,00000110B
59:            DB      00100000B,00111110B
60:            DB      00110000B,01111110B
61:            DB      00011111B,11111100B
62:            DB      00011111B,11111000B
63:            DB      00000111B,11000000B
64:            ;   GREEN PLANE DATA
65:            DB      00000000B,11000000B
66:            DB      01000000B,01110000B
67:            DB      00010000B,00111100B
68:            DB      10000000B,00011110B
69:            DB      00000000B,00011110B
70:            DB      00100000B,00011110B
71:            DB      00000000B,00010111B
72:            DB      00000000B,00011111B
73:            DB      00000000B,00111111B
74:            DB      00000000B,00011011B
75:            DB      01000000B,00000110B
76:            DB      00100000B,00111110B
77:            DB      00110000B,01111110B
78:            DB      00011111B,11111100B
79:            DB      00011111B,11111000B
80:            DB      00000111B,11000000B
```

……………キャラクタの縦幅, 横幅

末尾がBの数値は, 2進数を表す
0と1がドットのON/OFFに, 対応しているので
グラフィックパターンを表現しやすい
このデータを変更すれば, キャラクタの形を
変えることができる

青プレーンのデータ

赤プレーンのデータ

緑プレーンのデータ

```
 81: ;
 82: ;          KEYBOARD INTERRUPT ROUTINE ……………キーボード割り込み処理ルーチン
 83: KEYINT  PROC
 84:         CMP     CS:COUNT,Ø   } カウンタをチェック
 85:         JA      KEY_EXIT     } 0ならばキャラクタを表示中なので消去する
 86:         ;
 87:         PUSH    AX  ┐
 88:         PUSH    BX  │
 89:         PUSH    CX  │
 90:         PUSH    DX  │ レジスタを保存する
 91:         PUSH    SI  │
 92:         PUSH    DI  │
 93:         PUSH    DS  │
 94:         PUSH    ES  ┘
 95:         ;
 96:         MOV     AH,ØCH  } テキスト画面の表示を再開する
 97:         INT     18H     } ROM BIOS呼び出し
 98:         CALL    PUTMARK ……………キャラクタを消去する
 99:         ;
100:         POP     ES  ┐
101:         POP     DS  │
102:         POP     DI  │
103:         POP     SI  │ レジスタの復帰
104:         POP     DX  │
105:         POP     CX  │
106:         POP     BX  │
107:         POP     AX  ┘
108: KEY_EXIT:
109:         MOV     CS:COUNT,RTIMER ……………カウンタをリセット
110:         JMP     DWORD PTR CS:KEYVEC ……本来のキーボード割り込みルーチンへジャンプ.
                                          この命令はアドレスKEYVECのメモリに格納され
                                          ているデータの指すアドレスへジャンプする命
                                          令である
111: KEYINT  ENDP
112:
113:
114: VSYNC   PROC ……………CRTC(VSYNC)割り込み処理ルーチン
115:         PUSH    AX  ┐
116:         PUSH    BX  │
117:         PUSH    CX  │
118:         PUSH    DX  │ レジスタの保存
119:         PUSH    SI  │
120:         PUSH    DI  │
121:         PUSH    DS  │
122:         PUSH    ES  ┘
123:         ;
124:         CMP     CS:COUNT,Ø    } カウンタをチェック
125:         JE      VSYNC_ANIME   } 0ならばアニメーション中
126:         DEC     CS:COUNT      } カウンタをデクリメント
127:         JA      VSYNC_EXIT    } 0になればアニメーション開始
128:         ;
129:         MOV     AH,ØDH  } テキスト画面の表示を停止する
130:         INT     18H     } ROM BIOS呼び出し
131:         ;
132:         JMP     ANIME_START ……………アニメーションを開始する
133: VSYNC_ANIME:
```

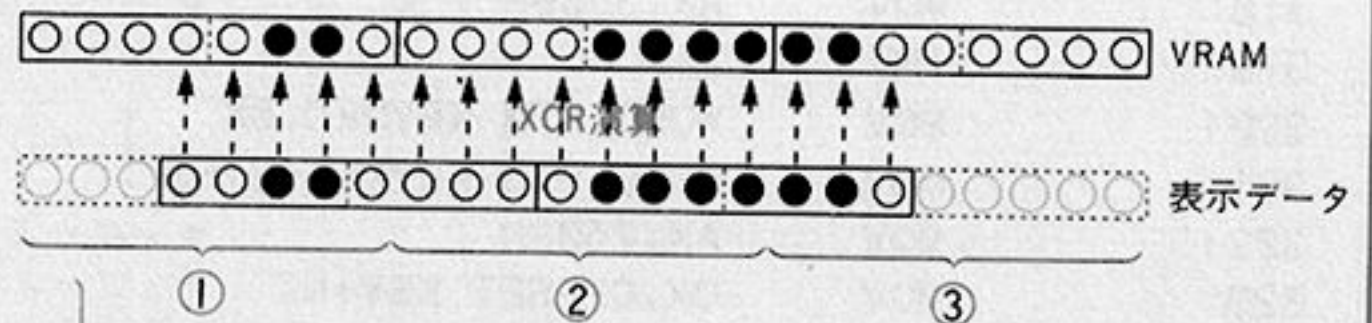
初期化ルーチンを除きデータラベルの参照に，すべて「CS:」というセグメントオーバーライドプリフィックスが付いている点に注目．割り込み処理ルーチンでは，CSレジスタ以外のレジスタの値はどうなっているかわからないからである

```
134:            CALL    PUTMARK ……………キャラクタを消去する
135: ANIME_START:
136:            CALL    ANIME ……………キャラクタを移動し再表示する
137: VSYNC_EXIT:
138:            MOV     AL,20H  ┐
139:            OUT     00H,AL  ┘割り込みコントローラにEOIを発行する
140:            OUT     64H,AL ……………CRTCから割り込みがかかるように
141:            ;                        CRTCをリセットする
142:            POP     ES ┐
143:            POP     DS │
144:            POP     DI │
145:            POP     SI │
146:            POP     DX │レジスタの復帰
147:            POP     CX │
148:            POP     BX │
149:            POP     AX ┘
150:            IRET ……………割り込みルーチンを終了する
151: VSYNC      ENDP
152:
153: RND        PROC ……………乱数発生ルーチン
154:            MOV     AH,0   ┐タイマ/カウンタ(常に数をカウントしているハードウェア)から,
155:            IN      AL,73H ┘カウンタ値を読み出す
156:            AND     AL,3   ┐(カウンタ mod 3)+1を乱数として利用する
157:            INC     AL     ┘1,2,3,4の4種類のデータが得られる
158:            RET
159: RND        ENDP
160:
161: ANIME      PROC ……………アニメーション処理ルーチン
162:            MOV     CX,CS:X       ┐
163:            ADD     CX,CS:DELTAX  ┘キャラクタのX座標を移動させる
164:            CMP     CX,MIN_X              ┐
165:            JG      CHK_R                 │
166:            CALL    RND                   │
167:            MOV     CS:DELTAX,AX          │
168:            MOV     CX,CS:X               │
169:            JMP     CHK_Y                 │
170: CHK_R:                                   │移動範囲を超えていたら,
171:            CMP     CX,MAX_X-WIDTH_X*8    │移動方向を反転させ
172:            JL      CHK_Y                 │移動量を乱数で決定する
173:            CALL    RND                   │
174:            NEG     AX                    │
175:            MOV     CS:DELTAX,AX          │
176:            MOV     CX,CS:X               │
177: CHK_Y:                                   │
178:            MOV     CS:X,CX               ┘
179:            MOV     CX,CS:Y       ┐
180:            ADD     CX,CS:DELTAY  ┘キャラクタのY座標を移動させる
181:            CMP     CX,MIN_Y      ┐
182:            JG      CHK_D         │
183:            CALL    RND           │
184:            MOV     CS:DELTAY,AX  │
185:            MOV     CX,CS:Y       │
186:            JMP     MOVE          │
```

```
187: CHK_D:                                          移動範囲を超えていたら,
188:          CMP     CX,MAX_Y-WIDTH_Y               移動方向を反転させ
189:          JL      MOVE                           移動量を乱数で決定する
190:          CALL    RND
191:          NEG     AX
192:          MOV     CS:DELTAY,AX
193:          MOV     CX,CS:Y
194: MOVE:
195:          MOV     CS:Y,CX
196:          CALL    PUTMARK ……………新しい位置にキャラクタを表示する
197:          RET
198: ANIME    ENDP
199:
200: PUTMARK PROC ……………キャラクタ表示ルーチン
201:          MOV     CX,CS:X
202:          MOV     DX,CS:Y                        パラメータをレジスタにセット
203:          MOV     SI,OFFSET PATTERN_BUF
204:          MOV     AX,CS                          DSレジスタの値をCSレジスタと同じにする
205:          MOV     DS,AX
206:          CALL    PUT ……………表示ルーチンをコール
207:          RET
208: PUTMARK ENDP
209:
210: ;        SEGMENT ADDRESS OF VRAM FOR NEC PC-9801
211: B_PLANE DW       0A800H                         PC-9801のグラフィックVRAMの
212: R_PLANE DW       0B000H                         セグメントアドレス
213: G_PLANE DW       0B800H
214:
215: GETADR   PROC ……………画面上の座標からグラフィックVRAMでの
                          オフセットアドレスを計算するルーチン
216: ;        INPUT
217: ;         CX : X                                入力パラメータ
218: ;         DX : Y
219: ;        OUTPUT
220: ;         DI : VRAM ADDRESS                     計算結果
221: ;         CX : BIT ADDRESS
222:          MOV     AX,80
223:          MUL     DX
224:          MOV     DI,AX
225:          MOV     AX,CX                          DI=Y×80+X/8
226:          SHR     AX,1
227:          SHR     AX,1
228:          SHR     AX,1
229:          ADD     DI,AX
230:          AND     CX,07H ……………CX=X mod 8
231:          RET                    00000111BとANDをとることにより
232: GETADR   ENDP                   8で割った余りが得られる
233:
                          表示/消去ルーチン
234: PUT      PROC ……………一度表示した後, 同じパラメータで呼び出すと消去する
235: ;        INPUT
236: ;         CX : X
237: ;         DX : Y                                入力パラメータ
238: ;         DS:SI : IMAGE
```

```
239:          CALL    GETADR ··············VRAM上のアドレスを計算
240:          LODSW  }
241:          MOV     DX,AX }キャラクタの縦幅をDXレジスタにセット
242:          LODSW  }
243:          MOV     BX,AX }キャラクタの横幅をBXレジスタにセット
244:          ;
245:          MOV     ES,CS:B_PLANE }
246:          CALL    PUT_PLANE     }青プレーンの処理
247:          MOV     ES,CS:R_PLANE }
248:          CALL    PUT_PLANE     }赤プレーンの処理
249:          MOV     ES,CS:G_PLANE }
250:          CALL    PUT_PLANE     }緑プレーンの処理
251:          RET
252: PUT      ENDP
253:
254: PUT_PLANE        PROC ··············1つのプレーンについての処理ルーチン
255:          PUSH    DX }
256:          PUSH    DI }レジスタの保存
257: P_LOOPV:
258:          PUSH    DI
259:          PUSH    BX
260:          JCXZ    PUT_JUST ··············X座標が8の倍数なら専用ルーチンへ
261:          ; LEFT BYTE
262:          MOV     AL,[SI]      }
263:          SHR     AL,CL        }
264:          XOR     ES:[DI],AL   }左端バイトを表示──①
265:          INC     DI           }
266:          ;
267: PUT_M:
268:          DEC     BX
269:          JZ      PUT_R
270:          ; MID BYTES
271:          MOV     AH,[SI]      }
272:          MOV     AL,[SI+1]    }
273:          INC     SI           }
274:          SHR     AX,CL        }中間バイトを表示──②
275:          XOR     ES:[DI],AL   }
276:          INC     DI           }
277:          JMP SHORT PUT_M
278: PUT_R:
279:          MOV     AH,[SI]      }
280:          INC     SI           }
281:          XOR     AL,AL        }右端バイトを表示──③
282:          SHR     AX,CL        }
283:          XOR     ES:[DI],AL   }
284:          JMP     PUT_NEXTLINE
285: ;
286: PUT_JUST:
287:          PUSH    CX           }
288:          MOV     CX,BX        }
289: J_LOOP:                       }
290:          LODSB                }X座標が8の倍数のときの処理
291:          XOR     ES:[DI],AL   }
```

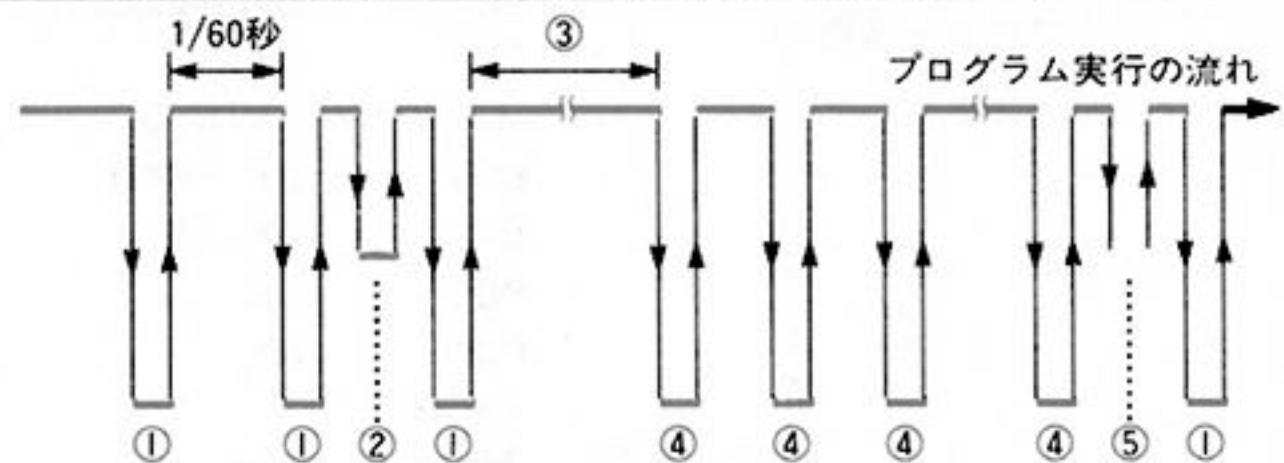
VRAM
XOR演算
表示データ
① ② ③

X XOR Y	0	1
0	0	1
1	1	0

(X XOR Y) XOR Y = X
同じデータで2回XOR演算すると，もとの値に戻ってしまう．
このことを利用して1つのルーチンで，表示と消去を行っている．

```
292:          INC     DI
293:          LOOP    J_LOOP
294:          POP     CX
295:          ;
296: PUT_NEXTLINE:
297:          POP     BX
298:          POP     DI
299:          ADD     DI,8Ø
3ØØ:          DEC     DX
3Ø1:          JNZ     P_LOOPV
3Ø2:          ;
3Ø3:          POP     DI
3Ø4:          POP     DX
3Ø5:          RET
3Ø6: PUT_PLANE        ENDP
3Ø7: ----------------------------------------------------------------
3Ø8: INIT:
3Ø9:          CALL    RND
31Ø:          SHL     AX,1
311:          SUB     AX,5
312:          MOV     DELTAX,AX
313:          CALL    RND
314:          SHL     AX,1
315:          SUB     AX,5
316:          MOV     DELTAY,AX
317:          ;
318:          MOV     AX,35Ø9H
319:          INT     21H
32Ø:          MOV     WORD PTR KEYVEC,BX
321:          MOV     WORD PTR KEYVEC+2,ES
322:          MOV     AX,25Ø9H
323:          MOV     DX,OFFSET KEYINT
324:          INT     21H
325:          MOV     AX,25ØAH
326:          MOV     DX,OFFSET VSYNC
327:          INT     21H
328:          ;
329:          CLI
33Ø:          IN      AL,Ø2H
331:          AND     AL,ØFBH
332:          OUT     Ø2H,AL
333:          STI
334:          OUT     64H,AL
335:          ;
336:          MOV     AH,42H
337:          MOV     CH,ØCØH
338:          INT     18H
339:          MOV     AH, 4ØH
34Ø:          INT     18H
341:          ;
342:          MOV     DX,OFFSET INIT
343:          SHR     DX,1
```

```
344:            SHR     DX,1
345:            SHR     DX,1
346:            SHR     DX,1
347:            INC     DX
348:            MOV     AX,3100H
349:            INT     21H
350: ;
351: CODE       ENDS
352:            END START
```

常駐終了のファンクションコールにより初期化ルーチン(ラベルINIT以下)を除いた部分をメモリ中に常駐させたままプログラムを終了する
常駐部分のメモリサイズをパラグラフ数として求める(INITのアドレス/16+1)

スクリーン・セーバー・プログラムの仕組み

①CRTC割り込みルーチンでカウンタを減らす
②キーボード割り込みルーチンでカウンタを初期値に戻す.
③一定時間キーボードが押されなければ, カウンタが0になる. そこでテキスト画面の表示を停止する.
④カウンタが0の間, CRTC割り込みルーチンでアニメーションを表示する.
⑤キーボード割り込みルーチンで, カウンタを初期値に戻し, アニメーションを中止し, テキスト画面の表示を再開する.

リスト 6-8　FMR-60/70 用差分リスト

```
7行目を置換 ──────────── RTIMER   EQU     100*TIMER
9行目を置換 ──────────── MAX_X    EQU     1120
                       ┌ MAX_Y    EQU     750
                       │ UP_LIM   EQU     320
                       │ SEL_PLN  MACRO   R_S,G_S
                       │          PUSH    AX
                       │          PUSH    DX
                       │          MOV     AX,R_S
         11行目を置換   │          MOV     DX,402H
                       │          OUT     DX,AX
                       │          MOV     AX,G_S
                       │          MOV     DX,404H
                       │          OUT     DX,AX
                       │          POP     DX
                       │          POP     AX
                       └          ENDM
                       ┌ PAL_TBL  DW      30H
                       │          DB      0,0,0,0,0,0,1,0,0,0,0,0FFH
                       │          DB      2,0,0,0FFH,0,0,3,0,0,0FFH,0,0FFH
    18, 19行目の間に挿入 │          DB      4,0,0,0,0FFH,0,5,0,0,0,0FFH,0FFH
                       │          DB      6,0,0,0FFH,0FFH,0,7,0,0,0FFH,0FFH,0FFH
                       │ CAL_TBL  DB      10 DUP (0)
                       │ TIM_BLK  DW      0,0,0,0,1
                       └ INT_BLK  DB      6 DUP (0)
83行目を置換 ──────────── KEYINT   PROC    FAR
87行目から94行目を削除 ───
  96行目から97行目を置換 ┌          MOV     AX,010AH
                       └          INT     91H
100行目から107行目を削除 ──
114行目を置換 ─────────── VSYNC    PROC    FAR
115行目から122行目を削除 ──
 129行目から130行目を置換 ┌         MOV     AX,0105H
                       └          INT     91H
138行目から149行目を削除 ──
150行目を置換 ───────────          RET
                       ┌          PUSH    DI
                       │          MOV     AH,01H
                       │          MOV     DI,OFFSET CAL_TBL
 154行目から157行目を置換 │          PUSH    CS
                       │          POP     DS
                       │          INT     96H
                       │          MOV     AX,DS:8[DI]
                       └          POP     DI
 211行目から213行目を置換 ┌ PLANE_U  DW      0C000H
                       └ PLANE_D  DW      0C000H + (140 * UP_LIM)/16
                       ┌          MOV     ES,PLANE_U
                       │          CMP     DX,UP_LIM
         222行目を置換  │          JB      CALC_OFFSET
                       │          MOV     ES,PLANE_D
                       │          SUB     DX,UP_LIM
```

```
CALC_OFFSET:
                MOV     AX,14Ø
245行目を置換 ── SEL_PLN 1,Ø
247行目を置換 ── SEL_PLN 2,1
249行目を置換 ── SEL_PLN 4,2
299行目を置換 ── ADD     DI,14Ø
                MOV     AH,Ø1H
                MOV     DL,Ø1H
                PUSH    CS
                POP     DS
                INT     ØAEH
                MOV     AX,DS:2[DI]
                MOV     WORD PTR CS:KEYVEC, AX
                MOV     AX,DS:4[DI]
                MOV     WORD PTR CS:KEYVEC+2, AX
                MOV     AH,ØØH
                MOV     DL,Ø1H
                PUSH    CS
                POP     DS
318行目から341行目を置換
                MOV     DI,OFFSET INT_BLK
                MOV     WORD PTR INT_BLK+2,OFFSET KEYINT
                MOV     WORD PTR INT_BLK+4,CS
                INT     ØAEH
                MOV     AH,ØØH
                MOV     DI,OFFSET TIM_BLK
                MOV     WORD PTR TIM_BLK+2,OFFSET VSYNC
                MOV     WORD PTR TIM_BLK+4,CS
                PUSH    CS
                POP     DS
                INT     Ø97H
                MOV     AH,8ØH
                INT     92H
                MOV     AH,83H
                MOV     DI,OFFSET PAL_TBL
                INT     92H
```

6.3 デバイスドライバ

起動時にデバイスドライバを組み込めることは，MS-DOSの大きな特徴です．システムを作り直すことなくCONFIG.SYSファイルに登録するだけで，新たな周辺機器などをシステムに組み込むことができます．みなさんもこの機能を利用して，かな漢字変換フロントエンドプロセッサやRAMディスクなどを組み込んでいることでしょう．

デバイスドライバの構造や仕様は公開されており，誰でも作ることができます．別に何か機器を自作して，それをシステムに組み込むなどという場合でなくてもちょっとしたアイデアさえあればシステムに便利な機能を追加することができます．グラフィックVRAMを利用したRAMディスクなどはその好例です．本節では，5.7章で作成したローマ字カナ変換プログラムを，デバイスドライバとしてシステムに組み込むことを考えます．

デバイスドライバの構造は，実行型ファイルの構造とは異なる特殊なものなので，高級言語で記述することは困難です．アセンブラならばオブジェクトプログラムの構造を自由に構築することができるので，デバイスドライバの記述はMASMに適した題材といえるでしょう．

デバイスドライバについてのくわしい解説は，「応用MS-DOS」(アスキー出版局)などの他の参考書に任せるとして，ここでは，MASMでデバイスドライバを記述するポイントを解説することにしましょう．

デバイスドライバの概念

デバイスドライバは図6-16のような構造をしています．このなかのデバイスヘッダとは，デバイスの種類やデバイス名など，MS-DOSがデバイスを管理するために必要な情報を並べたものです．デバイスヘッダの部分だけは，必ずこの通りの順番でファイルの先頭になければなりません．したがってMASMでもそのように記述します．

デバイスとの入出力を行うルーチンは，「ストラテジルーチン」と「割り込みルーチン」という2つのルーチンに分けられています．デバイスドライバにおける割り込み処理ルーチンは，割り込みという名前がついていても，前節のハードウェア割り込みを処理する割り込み処理ルーチンとは意味が異なるので注意してください．MS-DOSはこれら2つのルーチンを，システムに組み込まれたサブルーチンとして扱い，デバイスに対する入出力指令を送ります．

図6-16 デバイスドライバの構造

MS-DOS からのデバイスドライバに対する指令は，まずストラテジルーチンに渡されます．ただし，このルーチンではその指令をとりあえず記録するだけですぐに終了しなければなりません．割り込み処理ルーチンは，ストラテジルーチンに続いて MS-DOS から呼び出され，ここで記録されている指令を実行します．

MS-DOS からデバイスドライバに渡される指令は「コマンドパケット」と呼ばれ，**図 6-17** のような構造をしています．このなかでリクエストヘッダという部分はどの指令でも同じですが，それ以外は指令の種類によって異なります．例題のプログラムでは OUTPUT という指令だけをサポートしているので，図 6-17 では OUTPUT 指令におけるコマンドパケットを示しています．

デバイスドライバに対する指令には，次ページの**表 6-2** のようなものがあります．この表には，ブロック型デバイス，キャラクタ型デバイスという用語がでてきますが，これはデバイスを大きく 2 種類に分けていることを表し

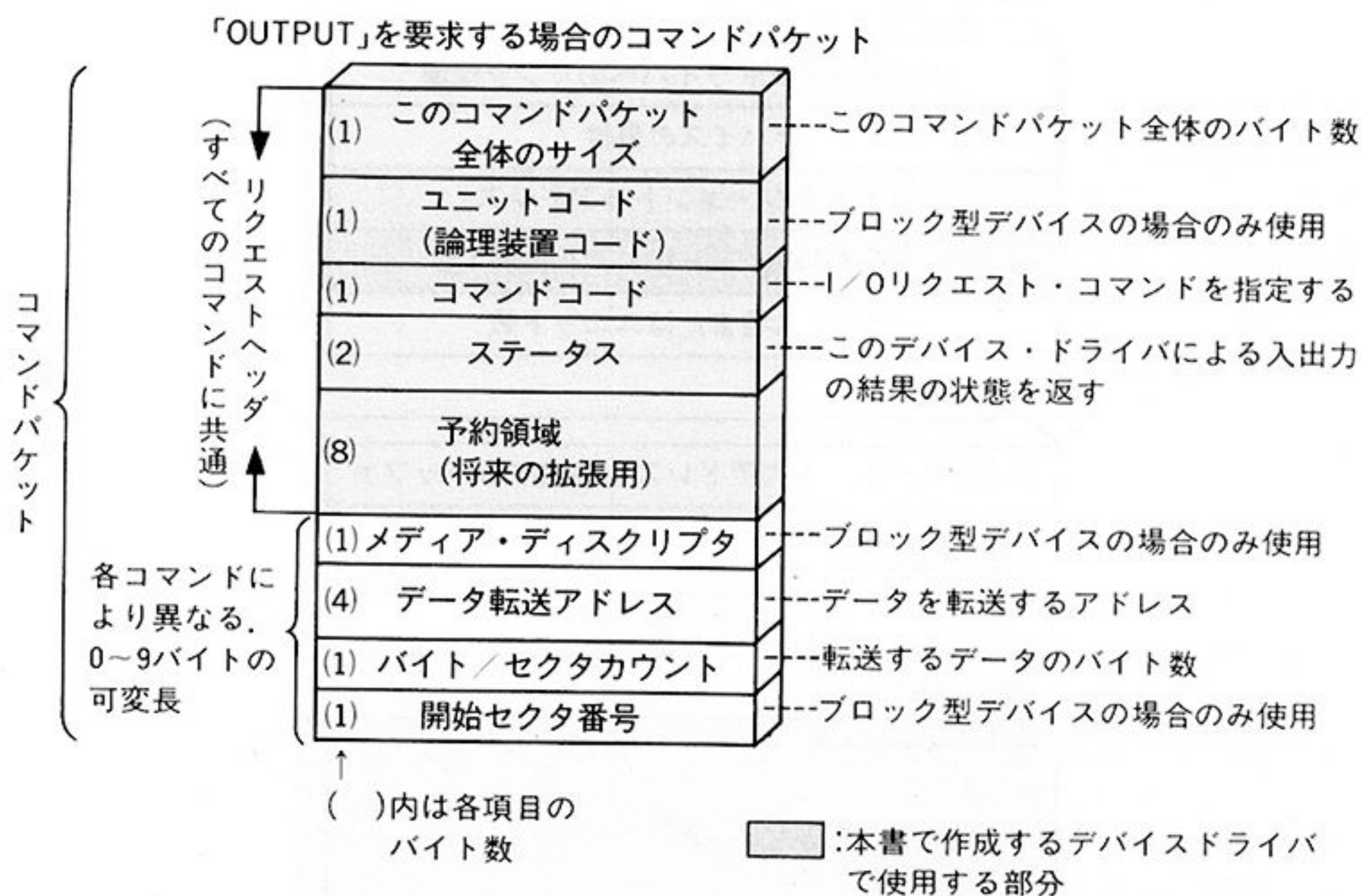



図 6-17　コマンドパケットの構造

ています．**ブロック型デバイス**は，ディスクドライブのようにドライブ名で区別するものを指します．**キャラクタ型デバイス**はそれ以外のもので，例題のデバイスドライバはこちらです．

コマンドコード	デバイス・ドライバの形式	コマンド	機能
0	B/C	INIT	デバイスドライバの初期化を行う
1	B	MEDIA CHECK	ディスクが交換されたかどうかを調べる
2	B	BUILD BPB	現在アクセスされているディスクに対応するBPBを作成する
3	B/C	IOCTL INPUT	デバイスドライバ自身からデータを入力する
4	B/C	INPUT	デバイスからデータを入力する(READ)
5	C	NON-DESTRUCTIVE INPUT NO WAIT	入力バッファの先頭の1バイトの内容を調べる
6	C	INPUT STATUS	入力バッファの内容が空かどうか調べる
7	C	INPUT FLUSH	入力バッファを空にする
8	B/C	OUTPUT	デバイスへデータを出力する(WRITE)
9	B/C	OUTPUT WITH VERIFY	デバイスへデータを出力するだけではなく，正しく出力できたかどうかの検査も行う
10	C	OUTPUT STATUS	出力バッファ内にデータが残っているかどうかを調べる
11	C	OUTPUT FLUSH	出力バッファの内容を空にする
12	B/C	IOCTL OUTPUT	デバイスドライバ自身へデータを渡す
13	B/C	DEVICE OPEN	デバイスのオープン．ただしデバイスドライバがOPEN/CLOSE/RMの機能を持つもののみ有効
14	B/C	DEVICE CLOSE	デバイスのクローズ．条件は13と同じ
15	B	REMOVABLE MEDIA	メディアが交換可能なデバイスかどうかを調べる．条件は13と同じ
16	C	OUTPUT UNTIL BUSY	デバイスがBUSY状態になるまで出力を続ける

B………ブロック型デバイスに対する機能
C………キャラクタ型デバイスに対する機能
B/C……共通の機能

(濃い網掛け)……本章で作成するデバイスドライバでサポートするコマンド，ただしコマンドコード9のベリファイの機能はなし

(薄い網掛け)……MS-DOS Ver3.1で拡張された機能

表6-2 I/Oリクエストコマンドの種類

デバイスドライバの利用方法

デバイスドライバにおける具体的な処理方法は，次項で取り上げるとして，できたデバイスをどのように利用できるのかをまず解説しましょう．

できあがったデバイスドライバのファイル名を「ROMAKANA.SYS」とします．このデバイスドライバをシステムに登録するには，CONFIG.SYS ファイルに次のように記述します．

```
DEVICE = ROMAKANA.SYS
```

システムをリセットして再起動すれば，このデバイスドライバがシステムに組み込まれます．もちろん，起動するディスクにROMAKANA.SYSをコピーしておくことはいうまでもありません．

キャラクタ型デバイスは，デバイスファイルとも呼ばれ，あたかも1つのファイルであるかのように扱うことができます．たとえば，「COPY CON CONFIG.SYS」とか，「COPY ROMAKANA.ASM PRN」というMS-DOSのコマンドを使ったことがあると思いますが，CONやPRNはそれぞれコンソール画面，プリンタというキャラクタデバイスの名前です．

同様にROMAKANA.SYSのデバイス名は，R2Kですから，

```
COPY TEGAMI.TXT R2K
```

のようにして利用することができます．この場合，ローマ字で書かれたTEGAMI.TXTがカナに変換されて表示されます．これでは5.7章で作成したフィルタプログラムとあまり変わりはありません．しかし，フィルタプログラムには不可能な利用法がR2Kにはあるのです．

通信プログラムなどの出力をR2Kにリダイレクトすると，リアルタイムでローマ字カナ変換を実現できます．プログラム実行途中の画面出力のローマ字が，その場でカナに変換されるのです．実行例を図6-18に示します．

```
A>cterm -b2400⏎ ……………通常の通信状態

 Note 151 ｼﾞｯｹﾝﾖｳ ﾉｰﾂ (junk.test)
[ RESPONSE:  1 of  1 ]
Title:t e s t  for  ro-ma ji
Date :  6:42am  5/22/88  From: pcs07520 (ﾀﾏ)

        「hajimete yomu M A S M」 wo yomou !

        minasann yorosiku .

                              pcs07520 tama

NOTES>>
```

```
A>cterm -b2400 >R2K⏎ ……………出力をR2Kデバイスに
                          リダイレクトした場合の
                          通信状態

 ﾉﾃ 151 ｼﾞｯｹﾝﾖｳ ﾉｰﾂ (jｳnk.ﾃst)
[ ﾚSPｵNｾ:   1 ｵf   1 ]
ﾁtlｴ:t ｴ s t  fｵr  ﾛｰﾏ jｲ
Dｱﾃ :  6:427m  5/22/88  Fﾛm: pcs07520 (ﾀﾏ)

        「ﾊjｲﾒﾃ ﾖﾑ M ｱ S M」 ｦ ﾖﾓｳ !   ローマ字がカナに変換され，
                                     表示されている
        ﾐﾅｻﾝ ﾖﾛｼｸ .

                              pcs07520 ﾀﾏ

ﾉﾃS>>
```

図6-18 R2Kデバイスドライバの利用例

＊この例は通信ソフトCTERMを使ってアスキーネットPCSにアクセスした例です．

サンプルプログラムーローマ字カナ変換デバイスドライバ

ローマ字カナ変換デバイスドライバは，3つのソースプログラムから構成されます(図6-19)．1つはデバイスヘッダの定義などを含むメインモジュールです(**リスト6-9**)．2つめは5章で作成したローマ字カナ変換モジュールをそのまま利用します(5章232ページの**リスト5-3**)．一度作った部品を再び利用できるというモジュール別プログラミングの利点を生かしましょう．

そして3つめはデバイスドライバの末尾のアドレスを知るためのダミープログラムです(**リスト6-10**)．このソースプログラムではラベルが1つ定義してあるだけで，マシン語コードは含まれません．デバイスドライバの初期化時に末尾アドレスが必要になるので，そのアドレスを知るためだけに存在するモジュールです．

アセンブル&リンクの例は，299ページの図6-20に示します．各モジュールをリンクする順序は，その図に示すとおりでなければなりません．また図に示すように，COMファイルを生成するのと同じ方法で，EXEヘッダを取り除きデバイスドライバファイルを作成します．

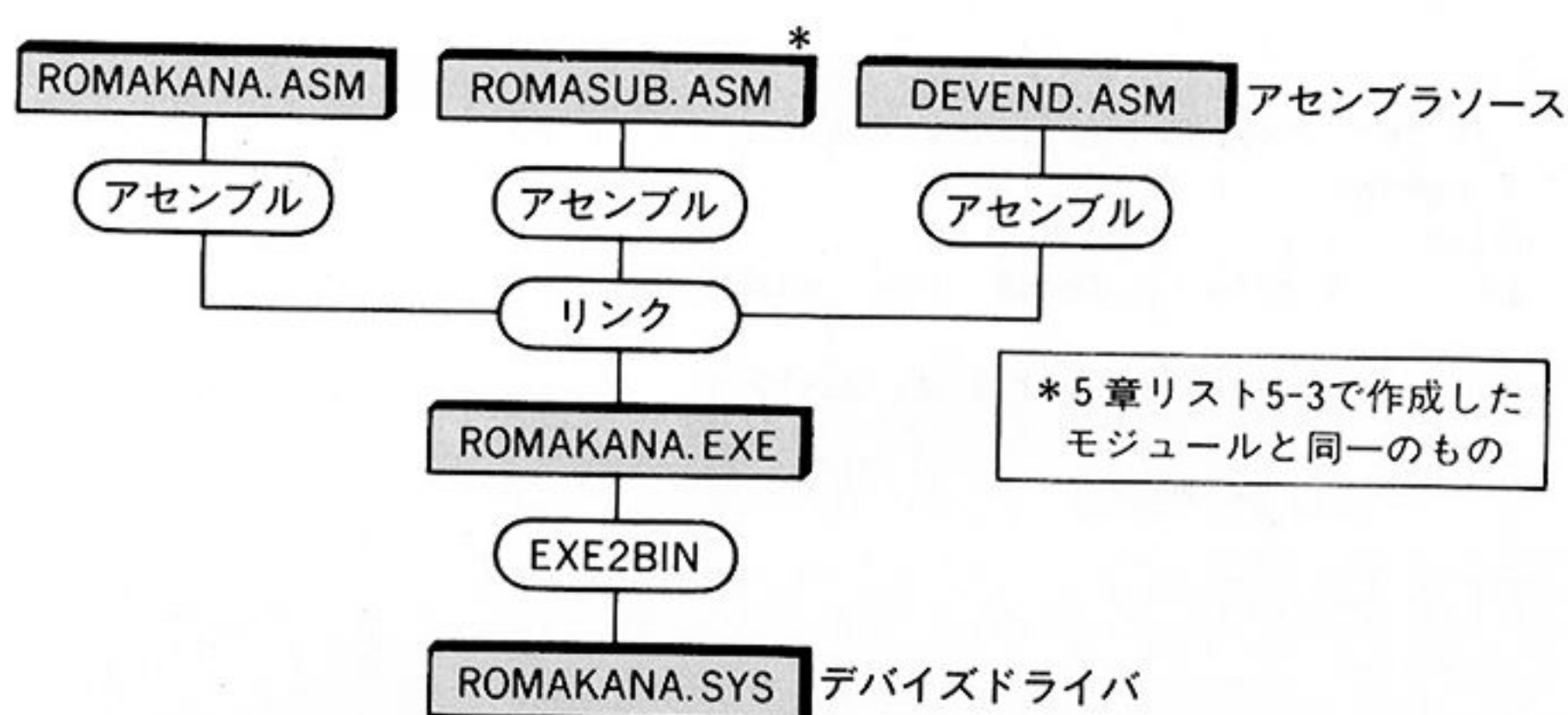



図6-19　ローマ字カナ変換デバイスドライバのプログラムの構成

リスト6-9 ローマ字カナ変換デバイスドライバ・メインプログラム ROMAKANA.ASM

```
        EXTRN   ROMAKANA:NEAR,CODEEND:NEAR ……………ローマ字カナ変換ルーチンは他の
                                                    モジュール内にある

CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE,DS:NOTHING,ES:NOTHING
                        デバイスドライバが呼び出されるときには,DS,ESレジスタの内容
COMMAND EQU     2       は不定である.このような場合,対応するセグメントとして
STATUS  EQU     3       NOTHINGを指定する
TRANS   EQU     14
BREAKADR EQU    14
COUNT   EQU     18
------------------------------------------------------------------------
 [デバイスヘッダ]………この部分だけは必ずこの順序でモジュール先頭になければならない
HEADER  DD      -1 ……モジュール内にデバイスドライバが1つしかない場合,ダブルワード(4バイト)をFFH
                                                                     で埋める
        DW      8000H ……………通常のキャラクタデバイスドライバを示すデバイス属性
        DW      STRATEGY ………ストラテジルーチンのオフセットアドレス
        DW      ENTRY ……………割り込みルーチンのオフセットアドレス
        DB      "R2K     " ……デバイスのファイル名(8文字,残りは空白で埋める)
------------------------------------------------------------------------
CMDTBL  DW      INIT
        DW      EXIT
        DW      EXIT
        DW      CMDERR
        DW      EXIT
        DW      EXIT
        DW      EXIT    各コマンドごとの処理ルーチンへのジャンプテーブル
        DW      EXIT
        DW      OUTPUT
        DW      OUTPUT
        DW      EXIT
        DW      EXIT
        DW      EXIT
PACKET  DD      ? ……………コマンドパケットのアドレスを格納する領域

SS_SAV  DW      ? ……………スタックセグメントレジスタ,スタックポインタの内容を保存する領域
SP_SAV  DW      ?
LSTACK  DW      80H DUP (?) ……………ローカルスタック領域
STACK_END: ……………………………… デバイスドライバを呼び出す側ではスタック領域をあまり
                                  用意していないので,独自にスタック領域を用意する
------------------------
STRATEGY        PROC    FAR         ……………スタックポインタの初期値を得るためのラベル
        MOV     WORD PTR PACKET,BX
        MOV     WORD PTR PACKET[2],ES  …… [ストラテジルーチン]
        RET                               コマンドパケットのアドレス(ES:BX)を保存して,
STRATEGY        ENDP                      すぐリターンする
------------------------------------------------------------------------
ENTRY   PROC    FAR                    …… [割り込みルーチン]
        PUSH    AX                        コマンドコードを調べて対応する処理を行なう
        MOV     CS:SS_SAV,SS  } スタック領域のアドレスを保存する
        MOV     CS:SP_SAV,SP  }
        MOV     AX,CS                  スタックセグメントレジスタ,スタックポインタにローカル
        MOV     SS,AX                  スタック領域のアドレスをセットする.8086CPUでは,SSレ
        MOV     SP,OFFSET STACK_END    ジスタに値を格納すると,次の1命令の実行が終わるまでハ
                                       ードウェア割り込みを受けつけない仕組みになっている
```

```
        PUSH    BX  ┐
        PUSH    CX  │
        PUSH    DX  │
        PUSH    SI  │
        PUSH    DI  ├ レジスタを保存する
        PUSH    BP  │
        PUSH    DS  │
        PUSH    ES  ┘

        LDS     BX,CS:PACKET ……………コマンドパケットのアドレスを取り出す
        MOV     CX,[BX+COUNT] …………コマンドパケットから転送バイト数を取り出す
        MOV     AL,[BX+COMMAND] ………コマンドパケットからコマンドコードを取り出す
        CBW
            ……………ALレジスタの最上位ビットの値をAHレジスタの全ビットにセットする命令.
                    1バイトのデータの符号を保ったまま1ワードに拡張することになる
        CMP     AX,12   ┐
        JA      CMDERR  ┘コマンドコードが12を超えていればエラー
        LES     DI,[BX+TRANS] ……………コマンドパケットから転送アドレスを取り出す
        SHL     AX,1        ┐
        MOV     SI,AX       ├ ジャンプテーブルを利用してコマンドごとの処理へジャンプ
        JMP     CS:CMDTBL[SI] ┘ ……………アドレス CS:CMDTBL[SI]のメモリの内容の指す
                                          アドレスにジャンプする命令
CMDERR:
        MOV     AX,81Ø3H  ┐
        JMP     ERREXIT   ┘ エラー番号をステータスコードとしてセットし,終了処理へ
-----------------------------------------------------------------------
INIT:
        LDS     BX,CS:PACKET            [初期化ルーチン]
        MOV     AX,OFFSET CODEEND       デバイスドライバの末尾のアドレスをコマンドパケットに
        MOV     [BX+BREAKADR],AX        格納して返す. このためにDEVENDモジュールが必要
        MOV     [BX+BREAKADR+2],CS
        JMP     EXIT
-----------------------------------------------------------------------
OUTPUT:
                                        [出力ルーチン]
        JCXZ    EXIT
OUTPUTLOOP:
        MOV     AL,ES:[DI]  ┐
        INC     DI          │
        PUSH    CX          │ 1バイトずつデータを取り出してプロシージャROMAKANAを呼び出す
        PUSH    DI          ├ ROMAKANAでは変換結果のバイト数をCXレジスタに,アドレスをES:
        PUSH    ES          │ BXに入れて返す
        CALL    ROMAKANA    ┘
        JCXZ    NOCHR
O_LOOP:
        MOV     AL,ES:[BX]  ┐ 変換結果を画面に出力する.デバイスドライバはMS-DOSシステム内部から
        INC     BX          │ 呼び出されるので,MS-DOSのシステムコールを呼び出すことはできない.
        INT     29H         ├ そこで割り込み番号29Hのソフトウェア割り込みでCONデバイスを直接呼び
        LOOP    O_LOOP      ┘ 出している. この機能は公式に公開されたものではないため,MS-DOSの将
                              来のバージョンでは使用できない可能性がある
NOCHR:
        POP     ES
        POP     DI
        POP     CX
        LOOP    OUTPUTLOOP
-----------------------------------------------------------------------
EXIT:
        MOV     AX,Ø1ØØH ……………正常終了を表すステータスコードをセット
ERREXIT:
```

```
        LDS     BX,CS:PACKET        } ステータスコードをコマンドパケットにセット
        MOV     [BX+STATUS],AX      }
        POP     ES     )
        POP     DS     |
        POP     BP     |
        POP     DI     |
        POP     SI     | レジスタの復帰
        POP     DX     |
        POP     CX     |
        POP     BX     )
        MOV     SS,CS:SS_SAV   } スタックセグメントレジスタ,スタックポインタの内容をもとに戻す
        MOV     SP,CS:SP_SAV   }
        POP     AX
        RET
ENTRY   ENDP

CODE    ENDS
        END
```

リスト 6-10 デバイスドライバ末尾用モジュール DEVEND.ASM

```
        ASSUME CS:CODE
        PUBLIC CODEEND
CODE    SEGMENT PUBLIC
CODEEND: ……………デバイスドライバの末尾アドレスを知るために必要なラベル
CODE    ENDS       このモジュールが末尾にくるようにリンクしなければならない
        END
```

```
A>MASM ROMAKANA;
Microsoft MACRO Assembler  Version 3.00
(C)Copyright Microsoft Corp 1981, 1983, 1984

Warning Severe
Errors  Errors
0       0
                      ┌─リンクは必ずこの順序でなければならない
A>LINK ROMAKANA+ROMASUB+DEVEND;
Microsoft (R) Overlay Linker  Version 3.51
Copyright (C) Microsoft Corp 1983, 1984, 1985, 1986.  All rights reserved.

Warning: no stack segment

A>EXE2BIN ROMAKANA ROMAKANA.SYS ……………EXE2BINコマンドでEXEヘッダを削除し,
                                        「SYS」という拡張子を持つデバイスドライバ
                                        を作成する
A>
```

図 6-20 デバイスドライバの作成手順

APPENDIX

80286CPUの機能とMS-OS/2

MASMコマンドの使い方

LINKコマンドの使い方

LIBコマンドの使い方

MASM擬似命令一覧

MS-DOS主要ファンクション一覧

80286CPUの機能とMS-OS/2

80286CPUは8086CPUを改良し，大幅に機能を追加したCPUです．さらにそれを32ビット化したものが80386CPUです．これらのCPUは8086CPUと互換性があり，高速の8086CPUとして使用することができます．

すでに多くのマシンに採用されている80286CPUは，高速性だけでなく非常に進んだ機能を取り入れているという点でも注目されています．そして今，MS-OS/2の登場により80286CPUの機能をフルに生かした環境が用意されています．

■ 80286CPUの動作モード

80286CPUは2つの動作モードを持っています．1つは8086CPUと互換性のある**リアルモード**で，もう1つは80286CPU独自の**プロテクトモード**です．

リアルモードは8086CPUとほぼ完全に互換なので，マシンを高速化するために利用されています．また，いくつかの便利なマシン語命令が追加されています．

プロテクトモードでは1Mバイトを超えるメモリ空間を扱うことができるほか，プロテクションやマルチタスクなど，大型コンピュータなみの機能を利用することができます．OS/2は，80286CPUのプロテクトモードにおけるこれらの機能を利用することにより，MS-DOSを大きく超えた環境を提供します．

■ プロテクトモードのアドレス変換機構

80286CPUとOS/2で実現される多くの機能を解説する前に，それを可能にする仕組みの1つである，80286CPUのプロテクトモードにおけるアドレス変換機能を解説しましょう．この仕組みを完全に理解する必要はありませんが，少しでも知っておくとOS/2の仕組みを容易に理解することができます．

プロテクトモードでもセグメントレジスタにセグメントアドレスを設定することで，対応するセグメントを指定します．しかしセグメントアドレスと物理アドレスとの対応は，リアルモードのように簡単ではありません．プロテクトモードのメモリアクセスは図1のような仕組みで行われます．

セグメントディスクリプタテーブルはメモリ上にある領域で，その先頭アドレスは，ディスクリプタテーブルレジスタ（GDTRおよびLDTR）で指定されます．この表はセグメントについての情報を並べたもので，1つの情報は「セグメント先頭の物理アドレス」，「セグメントの大きさ」，「属性」という3つのデータからなります．

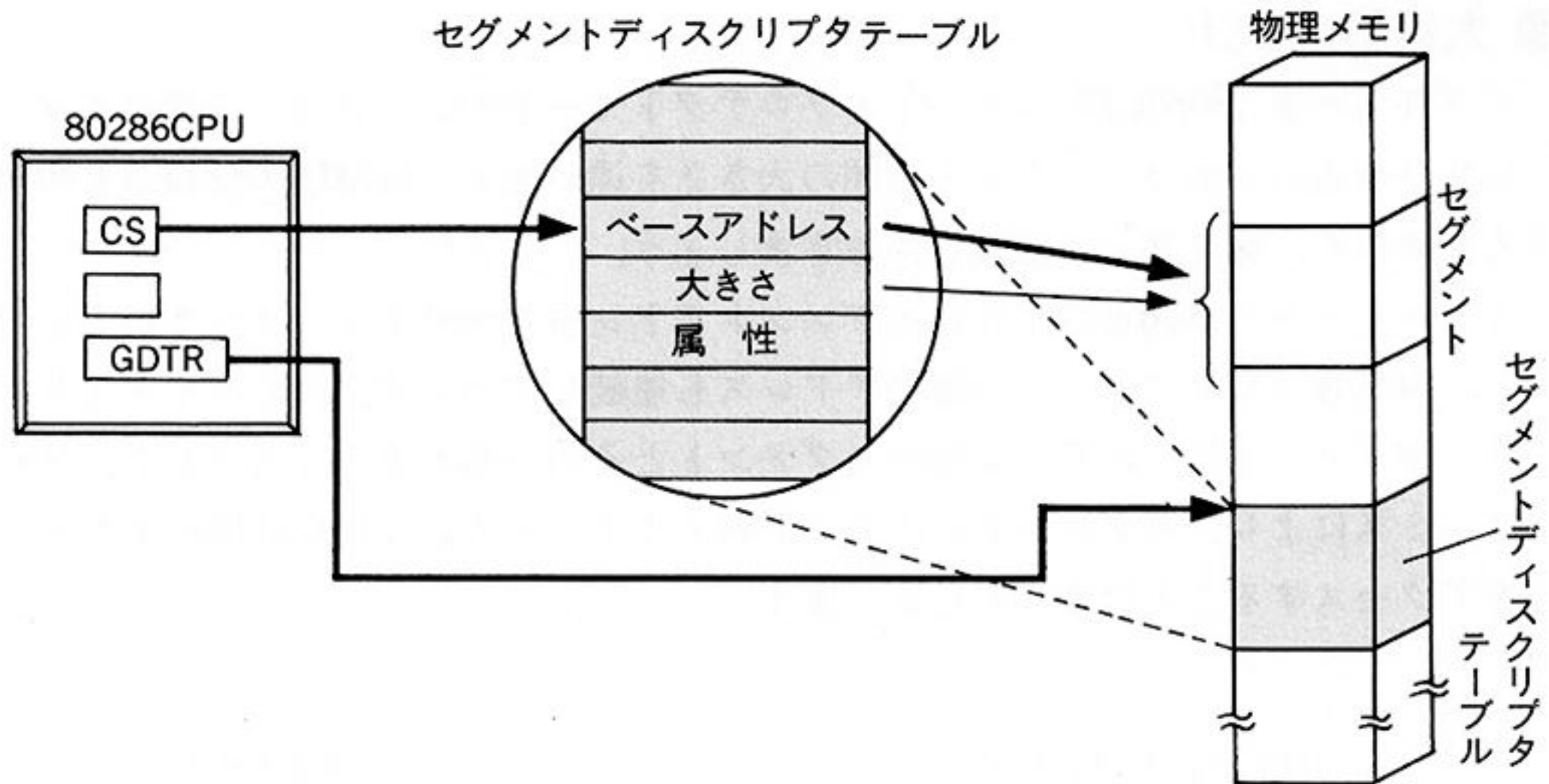



図1　プロテクトモードのメモリアクセス

セグメントアドレスは，この表において何番目に登録されているセグメントであるかを表すセグメント番号として扱われます．セグメント中のメモリをアクセスする際には，セグメントレジスタにセットされたセグメント番号から，表のなかのその番号のところにあるデータをもとに物理アドレスを求めます．つまり，セグメント先頭の物理アドレスにオフセットアドレスを加えたアドレスが目的のアドレスになります．

セグメントアドレスとオフセットアドレスによって表されるアドレスは，論理アドレスと呼ばれます．論理アドレスから物理アドレスへのアドレス変換はCPUに内蔵されたMMU（Memory Management Unit）というハードウェアによって自動的に行われます．論理アドレスと物理アドレスとの対応は，上の解説のようにディスクリプタテーブルによって管理されるので，この表にセグメントに関する情報をセットしておくだけでよいのです．

非常に複雑な仕組みのように感じられるかもしれませんが，私たちが作成するプログラムではこれまでの考え方となんら変わるところはありません．ディスクリプタテーブルの管理は，すべてOS/2システムの仕事です．私たちはこれまでどおり，セグメントを定義して，そのセグメントのセグメントアドレス（セグメント番号）をセグメントレジスタにセットするだけです．

アドレス変換機構（MMU）を用いて論理アドレスと物理アドレスを分離することにより，さまざまな機能を実現することができます．たとえば，これによりハードウェアで各種のチェックを行うことが可能になります．どのような機能を実現できるかを，その仕組みを含めてこのあと解説します．

■ 大容量メモリ

リアルモード（8086CPU モード）とプロテクトモードでは，メモリ空間のイメージに大きな違いがあり，アドレス空間の大きさも違います．MMU の仕組みを理解するためにも，図を使って解説しておきましょう．

リアルモードでは図 2 のようにセグメントアドレスと物理アドレスは 1 対 1 に対応し，隣り合ったセグメントは物理アドレスも連続しています．64K バイトよりも小さいセグメントは，実際には隣のセグメントとその一部が重なっています．プログラミスにより，セグメントの大きさを超えてアクセスした場合は隣のセグメントをアクセスすることになってしまいます．

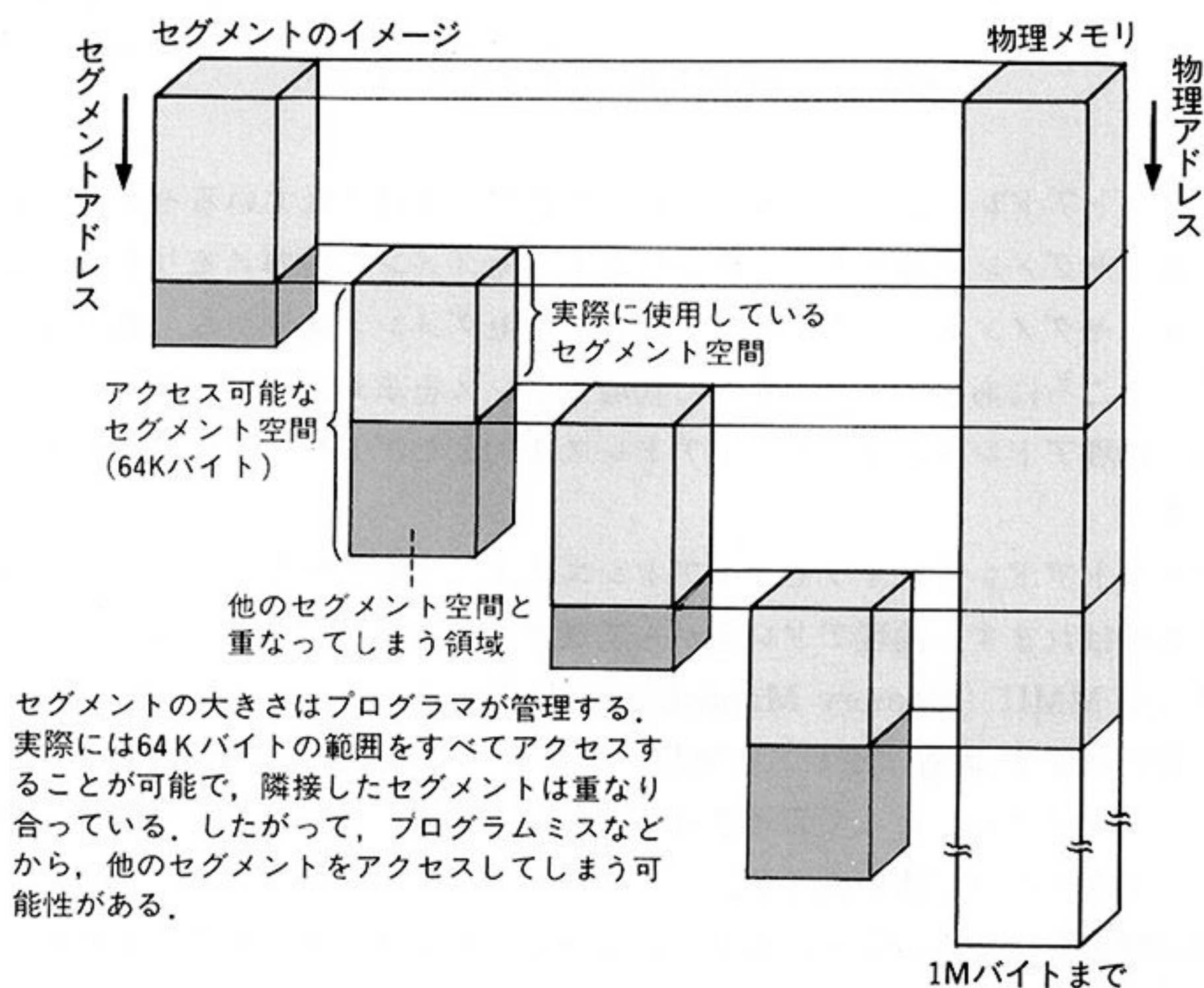



図 2　リアルモードでのメモリ管理

プロテクトモードでは 1 つ 1 つのセグメントはやはり 64K までの大きさしか持てませんが*，リアルモードのように隣り合うセグメントが重なりあうということがありません（図 3）．MMU の働きにより，隣り合うセグメントアドレスでもまったく独立した空間を持つことが可能です．このことは 80286CPU が 1M バイト以上のメモリをアクセスできることを意味します．

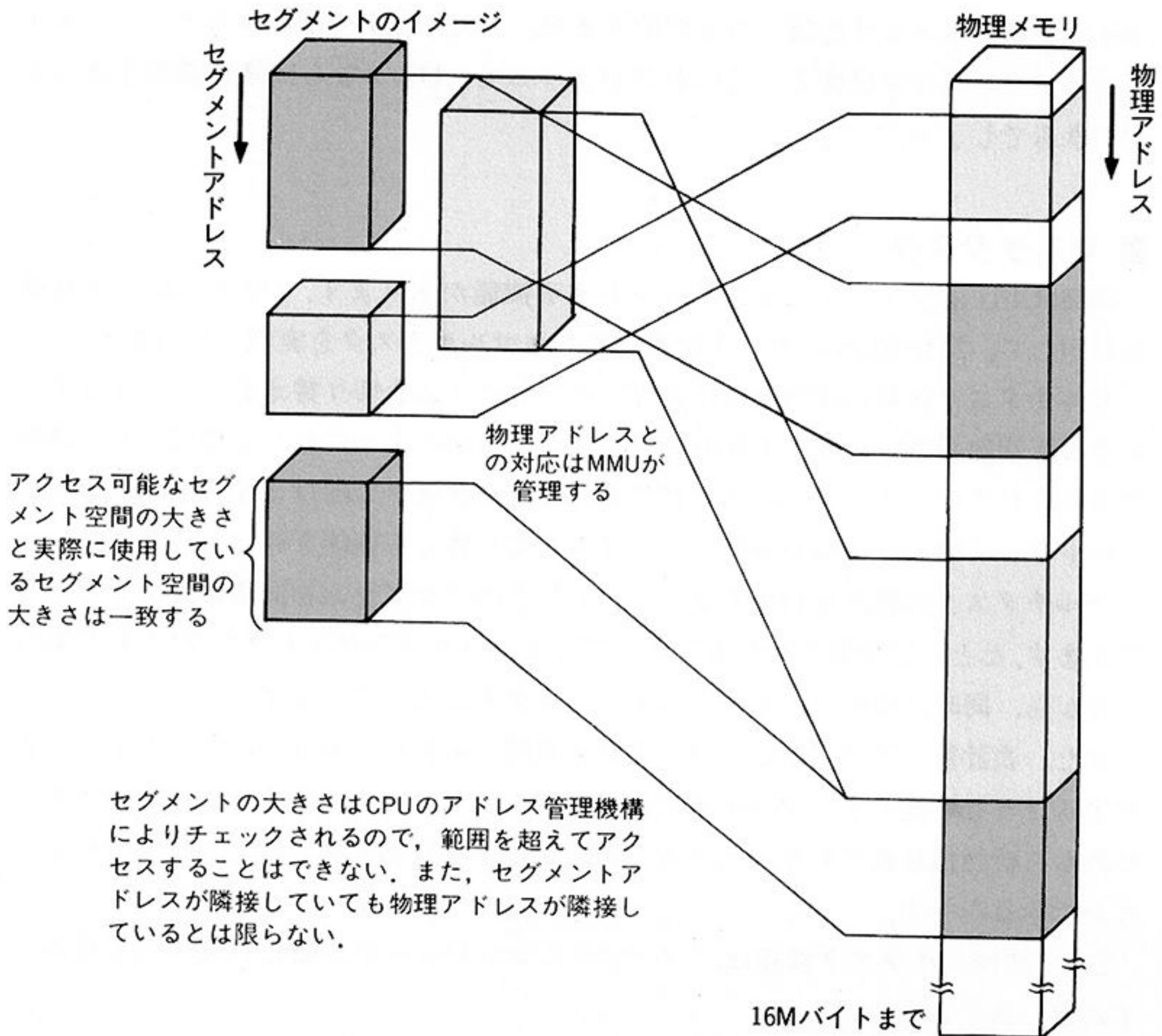



図3　プロテクトモードでのメモリ管理

リアルモードのメモリ空間は1Mバイトです．これ以上のメモリを増設しても，RAMディスクなどの利用法しかありません．これに対し80286CPUのプロテクトモードでは16Mバイトまでの物理メモリを直接扱うことができます*.

アプリケーションの高機能化により，必要とするメモリの量はどんどん大きくなっています．すでに，MS-DOSで扱えるメモリでは足りないというのが現状でしょう．このため80286CPU独自の機能を生かして多くのメモリを扱えるOSの登場が期待されていました．OS/2は80286CPUのプロテクトモードを利用して，1M

＊32ビットCPUである80386CPUのプロテクトモードでは，4G（ギガ，1G＝1000M）バイトまでの物理メモリを扱うことができる．また，セグメントの大きさが64Kバイトまでという制限はなく，すべてのメモリを1つのセグメントとして扱うことも可能．

バイトを超えるメモリを扱うことができます．メモリ容量の制限からオーバーレイなどのテクニックを必要としていたプログラムも，OS/2 ならば楽に開発することができるでしょう．

■ マルチタスク

80286CPU はマルチタスクをサポートする機能があります．OS/2 ではこの機能を利用して，MS-DOS との大きな違いであるマルチタスクを実現しています．

マルチタスクは短い時間ごとに実行するプログラムを切り替えることによって，あたかも複数のプログラムを同時に実行しているかのように見せる機能です．同時に実行されている1つ1つのプログラムのことを**タスク**と呼びます．80286CPU はハードウェア的にごく短い時間で，タクスを切り替える機能を持っています．

マルチタスクの機能を利用すると，いくつものプログラムを同時に動かすことができます．たとえば時間のかかるアセンブルをバックグラウンドタスク*として実行しながら，同時に他のファイルをエディットすることができます．

また，表計算のプログラムとワープロを同時に実行し，お互いにデータをやりとりすることも可能です．マルチタスクの利用により，いろいろなソフトウェアをはじめから統合化されていたシステムであるかのように切り替えながら使い分けることができるのです．

OS/2 のマルチタスク機能は，パーソナルコンピュータの新しい使い方を生み出すに違いありません．

■ MS-DOS 互換モード

OS/2 では 80286CPU のリアルモードを利用して，MS-DOS のプログラムを実行させることができます．MS-DOS 用に開発されたプログラムはかなりの数にのぼり，MS-DOS ユーザーの財産といってもよいでしょう．OS/2 はその財産を受け継ぎながら，かつ新しい機能を実現しているのです．

MS-DOS 用プログラムは，OS/2 の1つのタスクとして，同時に1つだけ実行させることができます．ただし，すべての MS-DOS 用プログラムが OS/2 でも実行できるわけではありません．OS/2 では次に解説するプロテクション機能によりハードウェアを直接操作することができませんが，MS-DOS 用プログラムはハードウェ

---

*キーボードからの入力を割り当てられているタスクをフォアグラウンドタスクと呼び，それ以外のタスクをバックグラウンドタスクと呼ぶ．バックグラウンドタスクはキーボード入力を必要としなければ，フォアグラウンドのタスクと並列に実行される．キーボード入力を待つバックグラウンドタスクは止まってしまうが，フォアグラウンドタスクと入れ替えて実行を再開することもできる．

アを直接操作するものが少なくありません．このためMS-DOS用プログラムは，ある制限のもとにハードウェアを直接操作することを許可されています．この制限に違反するプログラムはOS/2では動かないのです．

また，OS/2用プログラムはバックグラウンドでも動き続けますが，MS-DOS用プログラムはバックグラウンドでは止まってしまいます．これはVRAMに直接書き込みをするプログラムがフォアグラウンドタスクの画面を壊してしまうかもしれないからです．

制限付きながらもMS-DOSのプログラムが動作することは歓迎すべきことであり，OS/2用のプログラムが利用できるようになるまで，これまで使っていたすぐれたプログラムを利用していくことができるでしょう．

■ プロテクション機能

80286CPU独自の動作モードはプロテクトモード（保護モード）と呼ばれていますが，その名のとおりハードウェア的な保護機能が働きます．この機能により，OS/2ではプログラムミスによってシステム領域を破壊してしまったり，ハングアップしたりすることがありません．

たとえば，セグメントの大きさはディスクリプタテーブルにセットされており，これはマシン語命令の実行のたびにチェックされます．セグメントの大きさを超えてメモリをアクセスしようとすると，割り込みが発生しプログラムの実行は中断されます．この機能によりプログラムミスの発見が容易になります．

さらにセグメントごとに，「実行専用」，「読み出し可能」，「書き込み可能」などの属性を設定することができます．これらの属性もマシン語命令の実行のたびにチェックされます．違反すると割り込みが発生し，プログラムの実行は中断します．この機能により，プログラムミスからコード部分に書き込みを行ってしまい暴走するということはなくなります．

また，ディスクリプタテーブルはタスクごとに独立に存在し，1つのタスクから他のタスクのセグメントをアクセスすることはできません．これはマルチタスクにおいては重要な機能で，1つ1つのタスクは他のタスクに影響を与えないことがハードウェア的に保証されます．

I/Oポートへの入出力やVRAMのアクセスは，OS/2のシステムタスクにのみ許可されます．それ以外のユーザータスクからアクセスしようとすると割り込みが発生し，プログラムの実行は中断します．ハードウェアのアクセスはすべてOS/2の管理下におかれ，OS/2へのシステムコールを通してのみアクセスすることが可能です．このことにより，ハードウェアの操作方法はどの機種でも統一され，機種に依

存しないプログラムを作成することができます*.

MS-DOSではプログラムミスから暴走することは避けられませんでしたが，OS/2では以上のようなプロテクション機能により，ユーザープログラムが暴走しても必ずシステムに戻ることができます．このような高信頼性，耐障害性のおかげで安心してプログラムを開発したり，アプリケーションソフトを利用することができるでしょう．

なお，OS/2のMS-DOS互換モードはリアルモードで実行されるので，これらのプロテクションは働きません．したがって暴走したりすると，バックグラウンドで動作しているOS/2タスクを含めて回復不能になってしまいます．

■ 仮想記憶

アドレス変換機構は，「仮想記憶」を可能にします．仮想記憶とは，実際に搭載されているメモリの量よりも多くのメモリが存在するかのように見せる仕組みのことです．搭載されているメモリの量を気にすることなく，あたかもたくさんのメモリがあるかのようにプログラムを書くことができます．

仮想記憶は次のような方法で実現されます．物理メモリよりも多くのメモリが必要になったら，一部のメモリの内容をディスクにセーブします（スワップアウト）．そうして空いたメモリを利用します．ディスクにセーブされた部分のメモリの内容が必要となると，その時点でディスクからメモリに戻します（図4）．

OS/2では，プログラムをメモリからロードするときやプログラムからメモリを要求されたときなどに，物理メモリの一部を必要な分だけセグメントとして割り当てていきます．そして物理メモリをすべて割り当ててしまった状態でさらにメモリが必要になると，あまり使われていないセグメント**の内容をディスクへセーブします．その際にディスクリプタテーブルの「存在ビット」をクリアします．こうしてそのセグメントの占めていたメモリを，新たに必要となったプログラムに割り当てます．

このままプログラムが実行されていくと，さきほどディスクにセーブしてメモリ上には存在しなくなったデータが再び必要になります．このセグメントをアクセスしようとすると，存在ビットが0なので自動的に割り込みが発生します．この割り

---

*このことは50ページのコラムで解説した．

**プロテクトモードではセグメントをアクセスすると，ディスクリプタテーブルのアクセスビットがセットされる．OS/2はこのビットを定期的にチェックすることにより，セグメントの使用頻度を調べている．

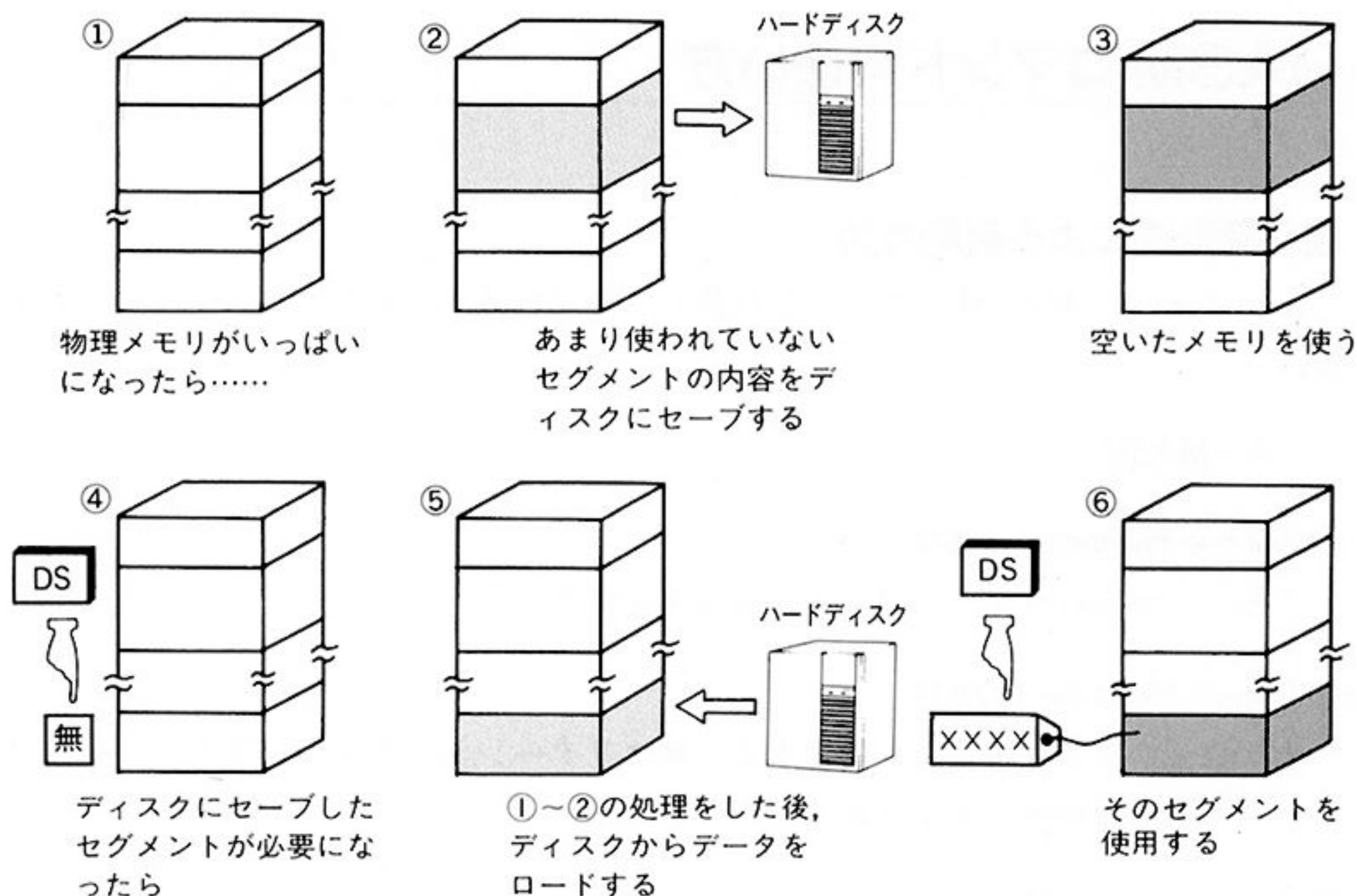



図4　仮想記憶の仕組み

込みによってOS/2のメモリ管理モジュールが呼び出され，ディスクにセーブしてあるセグメントの内容をメモリへロードします．そしてディスクリプタテーブルの存在ビットをセットし，割り込みから復帰します．そのセグメントをアクセスしたプログラムは，割り込みが発生したことも知らずにあたかもそのセグメントがずっとメモリ中にあったかのように実行を続けます．

このように仮想記憶を利用することにより，物理アドレス空間よりもはるかに大きな論理アドレス空間を実現することができるのです．80286CPUプロテクトモードの物理アドレス空間は16Mバイトですが，論理アドレス空間は実に1G（ギガ）バイトもあります*．

＊80386CPUのプロテクトモードでは，物理アドレス空間は4Gバイト，論理アドレス空間は64T（テラ，1T＝1000G）バイトである．

MASMコマンドの使い方

■対話形式による起動方法

以下のようにMASMコマンドを起動し，対話形式で必要なファイル名を入力する．

A>MASM ↵

① Source filename [.ASM]:

アセンブルを行うソースファイル名を指定する．

② Object filename [.OBJ]:

オブジェクトファイル名を指定する．単にリターンキーを入力すると，ソースファイル名が自動的に指定される（デフォルト）．

③ Source listing [NUL.LST]:

リスティングファイル名を指定する．必要ない場合は，リターンキーを押せばよい（デフォルト）．

④ Cross reference [NUL.CRF]:

クロスリファレンスファイル名を指定する．必要ない場合は，リターンキーを押せばよい（デフォルト）．

<指定例>

```
A>MASM ↵

  Source filename [.ASM]: main ↵
  Object filename [main.OBJ]: ↵
  Source listing [NUL.LST]: main/D ↵
  Cross reference [NUL.CRF]: refer ↵
```

- 入力ラインの終わりには，以降で示すオプションを付けることができる．
- ソースファイル名の指定以降では，「；」を入力するとそれ以後がデフォルトで処理される．

■コマンドラインによる起動方法

書式

MASM　ソースファイル名，[オブジェクトファイル名]，[リスティングファイル名]，[クロスリファレンスファイル名] [オプション]

- [] で囲まれた項目は省略可能
- オプションは，どの項目のあいだに入れてもよい
- ソースファイル名の指定以降は，「；」を指定して入力を終えることができる．

■ MASM コマンドのオプション

オプション	機　　能
/D	パス1のリスティングファイルを作成し，パス2のリスティングファイルと同時に出力する (例) masm test, , test/D;
/O	リスティングファイル中で8進基数による表示を行う (例) masm sample, , sample/O, sample
/ML†	シンボル名の大文字と小文字の区別をつける (例) masm sub1/ML;
/MX†	PUBLIC と EXTRN のシンボルに大文字と小文字の区別をつける (例) masm sub2/MX;
/MU††	PUBLIC と EXTRN のシンボル名を小文字から大文字に変換する（デフォルト） (例) masm sub3/MU;
/X	リスティングファイル中に，アセンブルされない偽の条件文リストも出力する (例) masm true, , true/X;
/R†	浮動小数点演算の数値演算プロセッサ用のコード生成する．8087, 80287 コプロセッサを備えたマシンのみ実行可能 (例) masm math1/R;
/E†	浮動小数点演算のエミュレートコードを生成する．浮動小数点演算用のライブラリをリンクする必要がある (例) masm math2/E;
/A†	オブジェクトファイルを出力する際，セグメントをセグメント名のアルファベット順に出力する (例) masm abc/A, ,abc;
/S††	オブジェクトファイルを出力する際，セグメントをソースファイルの出てくる順に出力する（デフォルト） (例) masm source/S;
/N††	リスティングファイル中の最後のテーブル（セグメント，シンボルの一覧など）をすべて削除する (例) masm source, ,/N;
/B<n>††	アセンブル時に，ソースファイルに割り当てるバッファサイズ n を指定する（デフォルトは 32K バイト）．バッファサイズをソースファイルより大きくすれば，メモリ上でアセンブルを行うことができ，処理スピードが向上する (例) masm test, ,/B16;

オプション	機　　　能
/C††	クロスリファレンスファイルを作成する (例) masm test/C;
/L††	リスティングファイルを作成する (例) masm sample/L;
/D<symbol>††	シンボルをヌルの文字列として定義する．ソースファイル中で，あたかもEQU擬似命令で定義したかのように使用できるので，IFDEF擬似命令などの条件アセンブルに活用できる (例) masm source, ,/DDEBUG;
/I<path>††	インクルードファイルを検索するサーチパスを指定する (例) masm romakana, ,/Ia:¥masm¥macro /If:¥lib;
/T††	警告やエラーなしにアセンブルが終了した場合，メッセージを出力しない (例) masm complete/T;
/V††	アセンブル終了時に，通常の情報に加えて処理された行数やシンボル数を表示する (例) masm error1/V;
/Z††	エラーを発生したソースラインの内容を表示する (例) masm error2, ,error/P;
/P††	80286CPUのプロテクトモードで問題となるコードをチェックする (例) masm back/P;

†は，MSAM Ver3.0で追加されたオプション
††は，MASM Ver4.0で追加されたオプション

LINKコマンドの使い方

■対話形式による起動方法

以下のようにLINKコマンドを起動し，対話形式で必要なファイル名を入力する．

A>LINK ↵

① Object Modules [.OBJ]:

オブジェクトファイル名を「＋」または空白で区切って，結合する順序に並べる．1行に書ききれない場合は，最後に「＋」を書くと，再度問い合わせてくれる．

② Run File [.EXE]:

生成される実行ファイルを指定する．単にリターンキーを入力すると，①の先頭で指定したオブジェクトファイル名が自動的に指定される（デフォルト）．

③ List File [NUL.MAP]:

マップファイル名を指定する．必要ない場合は，リターンキーを押せばよい（デフォルト）．

④ Libraries [.LIB]:

リンクするライブラリ名を①と同様に，「＋」または空白で区切って並べる．必要ない場合は，リターンキーを押せばよい（デフォルト）．

<指定例>

```
A>LINK ↵

  Object Modules [.OBJ]:main+sub1+sub2+ ↵
  Object Modules [.OBJ]:sub3+sub4/PAUSE ↵
  Run File [main.EXE]: test ↵
  List File [NUL.MAP]: ↵
  Libraries [.LIB]: ¥lib¥startup ↵
```

- 入力ラインの終わりには，以降で示すオプションを付けることができる．
- オブジェクトファイル名の指定以降では，「；」を入力するとそれ以後がデフォルトで処理される．

■コマンドラインによる起動方法

書式

LINK　オブジェクトファイル名 ..，[出力ファイル名]，[マップファイル名]，[ライブラリ名 ..] [オプション ..]

- [] で囲まれた項目は省略可能
- オプションは，どの項目のあいだに入れてもよい
- 複数のオブジェクトファイル名，およびライブラリ名を指定するときは，「＋」または空白で区切って並べる．
- オブジェクトファイル名の指定以降は，「；」を指定して入力を終えることができる．

■ LINK コマンドのオプション

オプション	省略形	機　　能
/CPARMAXALLOC:n†	/C:n	プログラムがメモリ中にロードされたときに必要な最大のパラグラフ数をセットする (例) link test, test /C:10;
/DOSSEG†	/DO	MS-DOS の順序付けの規約に従ってセグメントを配置する (例) link sample1 sample2 /DOSSEG;
/DSALLOCATE†	/D	"DGROUP"という名前のグループ内の最上位バイトに FFFFH を割り当てる．通常/HIGH オプションとともに用いられる (例) link sample /DSALLOCATE /HIGH;
/HIGH	/H	プログラムの開始アドレスを使用可能なメモリ中のできるだけ高いアドレスにセットする (例) link default+file /HIGH, file;
/LINENUNBERS	/LI	マップファイルに行番号情報を付加する (例) link file, file, file /LI;
/MAP	/M	パブリックシンボルのマップ情報をマップファイルに付加する．SYMDEB でデバッグする際には必要 (例) link file1.obj+file2.obj, file.exe, file.map/MAP;
/NODEFAULTLIBRARYSEARCH	/NOD	オブジェクトファイル内で検出される可能性のあるすべてのライブラリ名を無視する (例) link file.obj, file.exe, file.map, a:¥work¥math.lib/NOD;
/NOGROUPASSOCIATION†	/NOG	プログラムからグループを取り除く．このオプションは使用しない方がよい (例) link file/NOG;
/NOIGNORECASE†	/NOI	シンボル名の大文字と小文字を区別して扱う (例) link file.obj /NOIGNORECASE;
/OVERLAYINTERRUPT:n†	/O:n	オーバーレイローディングルーチンの割り込み番号を指定した値にセットする (例) link sample.obj, sample.exe/O:255;
/PAUSE	/P	実行可能ファイルを出力する前に休止する (例) link file.obj /PAUSE, file.exe;
/SEGMENTS:n†	/SE:n	プログラムのセグメント数の上限を指定する (例) link test1+test2, sample.exe, sample.map/SEGMENTS:20;
/STACK:n	/ST:n	スタックを指定されたバイト数にセットする (例) link file, file /STACK:0x2000;
/EXEPACK††	/E	実行可能ファイルをパックし，ロード時の再配置テーブルを最適化する．このオプションでパックしたファイルに対しては，SYMDEB を使用することができない (例) link lsize+bsize, size.exe/E
/HELP††	/HE	使用可能なオプション一覧を表示する (例) link /HELP

†は，LINK Ver2.40 で追加されたオプション
††は，LINK Ver3.00 で追加されたオプション

LIB コマンドの使い方

■対話形式による起動方法

以下のように LIB コマンドを起動し，対話形式で必要なファイル名を入力する．

A>LIB ⏎

① Library name:

作成／編集したいライブラリ名を入力する．新規のライブラリ名を入力すると，そのファイルを作成するかどうか聞いてくる．

② Operations:

以降で示す編集コマンドとオブジェクトファイル名を入力する．1行に書ききれない場合は，最後に「&」を書くと，再度問い合わせてくれる．

③ List file:

クロスリファレンスリスト名を指定する．必要のない場合は，リターンキーを押せばよい（デフォルト）．

④ Output library:

編集結果を出力するライブラリ名を入力する．リターンキーを押すと，①で指定したライブラリに結果を出力する．この場合，もとのファイルは「.BAK」という拡張子が付き，保存される（デフォルト）．

<指定例>

```
A>LIB ⏎

  Library name: int ⏎
  Operations: +RS-timer& ⏎
  Operations: -+stop+keyboard ⏎
  List file: int.lst ⏎
  Output library: ⏎
```

• オペレーションの指定以降では，「；」を入力するとそれ以後がデフォルトで処理される．

■コマンドラインによる起動方法

書式

LIB　ライブラリ名　[/PAGESIZE:n]　コマンド　[，リストファイル名，出力ライブラリ名]

- [ ] で囲まれた項目は省略可能
- PAGESIZE オプションは，ライブラリのページサイズ(デフォルトは16)を指定する．ページサイズの設定により，ライブラリのモジュールの配置（配列）を制御する．ページサイズが大きいほど，多くのモジュールを登録できるが，逆にむだな領域が増える．
- コマンドの指定以降は，「；」を指定して入力を終えることができる．

■ LIB コマンドの編集機能

<モジュールの追加>

機能　指定されたファイルのオブジェクトモジュールをライブラリに登録する

書式　＋<オブジェクトファイル名>

例　lib　graph　＋pset.obj　＋preset.obj ;

<モジュールの削除>

機能　ライブラリから指定したモジュールを削除する．

書式　－<モジュール名>

例　lib　graph　－line　－circle, graph.ref

<モジュールの置換>

機能　ライブラリから指定されたモジュールを削除した後，同名のオブジェクトモジュールを追加する

書式　－＋<モジュール名>

例　lib　graph　－＋a:￥obj￥paint;

<モジュールの抽出>

機能　ライブラリ中の指定されたモジュールと，同じ名前のオブジェクトモジュールを作成し，それにコピーする．

書式　＊<モジュール名>

例　lib　graph　＊ color ;

<モジュールの移動>

機能 ライブラリから指定されたモジュールを抜き出して，オブジェクトモジュールに出力する．

書式 －＊<モジュール名>

例 lib graph －＊ cls;

<ライブラリの連結>

機能 指定されたライブラリを連結して，1つのライブラリとする．

書式 ＋<ライブラリ名>

例 lib graph ＋text.lib

MASM疑似命令一覧

分類	疑似命令	機　　能	書　　式	参照ページ
メモリ疑似命令	ASSUME	セグメントレジスタがどの論理セグメント／グループを指しているかをアセンブラに告げる.	ASSUME 〈セグメントレジスタ〉:〈セグメント名〉[,…] NOTHING	88 132
	COMMENT	任意の大きさのコメントを入れる.	COMMENT 〈区切り記号〉 〈テキスト〉 〈区切り記号〉	
	DEFINE	変数を定義し, メモリを初期化する.	[〈変数名〉] {DB, DW, DD, DQ, DT} 〈式〉 [DUP, …]	73 76 77 78
	DUP	DEFINE 文のなかで, 式の繰り返しを定義する.	〈式 1〉 DUP (〈式 2〉 [, …])	75
	END	ソースプログラムを終了し, 実行開始アドレスを定義する.	END 〈式〉	71
	EQU	〈名前〉に値を割り振る.	〈名前〉 EQU 〈式〉	177 200
	=	〈名前〉に値を割り振る(再定義可能).	〈名前〉 = 〈式〉	255
	EXTRN	他のモジュール内で定義されている〈名前〉であることを宣言する.	EXTRN 〈名前〉:〈型〉 [, …]	223
	PUBLIC	〈名前〉が他のモジュールからも参照可能になるように指示する.	PUBLIC 〈名前〉 [, …]	222
	INCLUDE	他のアセンブラ・ソースファイルを挿入する.	INCLUDE 〈ファイル名〉	180
	LABEL	現在のロケーションに, 指定した〈型〉のラベルを生成する.	〈名前〉 LABEL 〈型〉	
	NAME	モジュール名を定義する.	NAME 〈モジュール名〉	
	PROC ~ ENDP	プロシージャを定義する.	〈プロシージャ名〉 PROC {[NEAR], FAR} ⋮ <プロシージャ> ENDP	181
	.RADIX	数値の基数を変更する(デフォルトは 10 進).	.RADIX 〈式〉	
	RECORD	ビット構造体を定義する.	〈レコード名〉 RECORD 〈フィールド名〉:〈ビット数〉 [=〈式〉] [, …]	
	GROUP	いくつかの論理セグメントを 1 つの物理セグメントに対応させる.	〈グループ名〉 GROUP 〈セグメント名〉 [, …]	139
	SEGMENT ~ ENDS	論理セグメントを定義する.	〈セグメント〉 SEGMENT [〈アラインメント〉] [〈コンバインタイプ〉] ['〈クラス名〉'] ⋮ 〈セグメント名〉 ENDS	87 127 226
	EVEN	ロケーション・カウンタが偶数になるように NOP 命令を挿入する.	EVEN	
	ORG	ロケーション・カウンタに〈式〉の値を代入する.	ORG 〈式〉	69

分類	疑似命令	機　　能	書　　式	参照ページ
メモリ疑似命令	STRUC ∫ ENDS	バイト構造体を定義する.	〈ストラクチャ名〉 STRUC ⋮ 〈フィールド名〉 DB / DW / DD / DQ / DT 〈式〉 [, …] ⋮ 〈ストラクチャ名〉 ENDS	
マクロ疑似命令	ENDM	MACRO, REPT, IRP, IRPC に対応してマクロ定義ブロックを閉じる.	ENDM	204
	EXITM	その時点でマクロ展開を中止する.	EXITM	
	IRP ∫ ENDM	IRP〜ENDM 間のブロックをすべての〈引数〉がダミーと置き換わるまで繰り返す.	IRP 〈ダミー〉, 〈[〈引数〉] [, …]〉 ⋮ ENDM	
	IRPC ∫ ENDM	IRPC〜ENDM 間のブロックをすべての〈文字列〉中の文字がダミーと置き換わるまで繰り返す.	IRPC 〈ダミー〉, 〈文字列〉 ⋮ ENDM	
	LOCAL	マクロ定義ブロック内でのローカルな名前を作る.	LOCAL 〈ダミー〉, [, …]	210
	MACRO ∫ ENDM	マクロを定義する.	〈マクロ名〉 MACRO [〈ダミー〉, …] ⋮ ENDM	204
	PURGE	指定したマクロ定義を削除する.	PURGE 〈マクロ名〉 [, …]	
	REPT ∫ ENDM	REPT〜ENDM 間のブロックを〈式〉で与えられる回数だけ繰り返す.	REPT 〈式〉 ⋮ ENDM	
特殊マクロ演算子		& 文字列または名前を連結する. < > 山形カッコ内のテキストを単一の引数として扱う. ;; あとに続くコメントをリスティングファイルに出力しない. ! 次のキャラクタを特殊キャラクタとして認識しない. % あとに続く式を現在の基数の数値に変換する.		

条件疑似命令	IF
	∫
	ELSE
	∫
	ENDIF
		IFxxxx によって指定される条件について, 満たされているときは IFxxxx〜ELSE (ELSE〜のないときには ENDIF まで), 満たされていないときには ELSE〜ENDIF (ELSE〜のないときには何も行わない) の間をアセンブルする.	IFxxxx [〈引数〉]
			⋮
			[ELSE]
			⋮
			ENDIF
				190
				194

	IF	式が 0 以外の値に評価された場合		
	IFE	式が 0 に評価された場合		
	IF1	アセンブラがパス 1 を実行中の場合		
	IF2	アセンブラがパス 2 を実行中の場合		
	IFDEF〈名前〉	〈名前〉が定義されているか, または EXTRN によって外部参照として宣言されている場合		

分類	疑似命令	機能	書式	参照ページ
条件疑似命令	IFNDEF〈名前〉	〈名前〉が未定義であり，外部参照としても宣言されていない場合		
	IFB〈[引数]〉	〈引数〉がブランク(何も与えられていない)または空白の(〈 〉のなかに何もはいっていない)場合		
	IFNB〈[引数]〉	〈引数〉がブランクかつ空白でない場合		
	IFIDN〈[引数1]〉,〈[引数2]〉	文字列〈引数1〉と〈引数2〉が同一である場合		
	IFDIF〈[引数1]〉,〈[引数2]〉	文字列〈引数1〉と〈引数2〉が異なっている場合		
リスティング疑似命令	PAGE	リスティングファイルの出力形式を指定する．改ページする．	PAGE [〈1ページの行数〉][,〈1行の文字数〉] [+]	
	TITLE	各ページに表示されるタイトルを指定する．	TITLE 〈文字列〉	
	SUBTTL	サブタイトルを指定する．	SUBTTL 〈文字列〉	
	%OUT	コンソールに〈文字列〉を表示する．	%OUT 〈文字列〉	
	.LIST	リスティングファイルへの出力を開始する(デフォルト)．	.LIST	
	.XLIST	リスティングファイルへの出力をすべて抑止する．	.XLIST	
	.SFCOND	条件式によりアセンブルされなかった部分のリスティングを抑止する．	.SFCOND	
	.LFCOND	条件式によりアセンブルされなかった部分のリスティングも出力する(デフォルト)．	.LFCOND	
	.TFCOND	現在の状態を反転する(デフォルトは／Xオプションによって決まる)．	.TFCOND	
	.XALL	マクロによって作成されるソースおよびオブジェクトコードをリスティングファイルに出力する(デフォルト)．	.XALL	
	.LALL	すべてのマクロを展開し，“；；”の前に付いたコメント以外のマクロ・テキストをリスティングファイルに出力する．	.LALL	
	.SALL	マクロによって作成されるすべてのテキストおよびオブジェクトコードのリスティングを抑止する．	.SALL	
	.CREF	クロスリファレンスファイルへの出力を開始する(デフォルト)．	.CREF	
	.XCREF	クロスリファレンスファイルへの出力を抑止する，あるいはクロスリファレンスファイルおよびシンボルテーブルから指定した変数を削除する．	.XCREF [〈名前〉][,〈名前〉]	
演算子	PTR	〈式〉の型(BYTE, WORD, DWORD, QWORD, TBYTE／NEAR, FAR)をオーバーライドする．	〈型〉 PTR 〈式〉	83 190
	:(セグメント・オーバーライド)	〈アドレス式〉の仮定されたセグメントをオーバーライドする．	〈セグメント・レジスタ〉 〈セグメント名〉 〈グループ名〉 :〈アドレス式〉	142

分類	疑似命令	機　能	書　式	参照ページ
演算子	SHORT	JMP 命令のラベルに対して使用し，SHORT ジャンプのコードを生成するようアセンブラに指示する．	SHORT 〈ラベル〉	
	THIS	指定した〈型〉の変数およびラベルを生成する．	THIS 〈型〉	
	HIGH	〈式〉で与えられた 16 ビット値の上位 8 ビットを分離する．	HIGH 〈式〉	
	LOW	〈式〉で与えられた 16 ビット値の下位 8 ビットを分離する．	LOW 〈式〉	
	SEG	変数またはラベルなどの，定義されたセグメントのセグメントアドレスを返す．	SEG 〈式〉	
	OFFSET	変数またはラベルなどの，定義された論理セグメントのなかでのオフセットアドレスを返す．	OFFSET 〈式〉	82 152
	TYPE	変数またはラベルなどのそれぞれ型属性を次に示す状態に従って返す． 変数の場合 BYTE =1 WORD =2 DWORD =4 QWORD =8 TBYTE =10 STRUC =STRUC によって割り当てられたバイト数 ラベルの場合 NEAR =FFFFH(−1) FAR =FFFEH(−2)	TYPE 〈式〉	
	.TYPE	〈式〉のモードおよび外部参照であるか否かを次に示す状態に従って返す． ビット 0，1：0 の場合，絶対モード(数値) 1 の場合，プログラム関連モード 2 の場合，データ関連モード ビット 5 ：1 の場合，モジュール内で定義されている． ビット 7 ：1 の場合，外部参照が含まれる．	.TYPE 〈式〉	
	LENGTH	変数のサイズを，定義された型を単位として返す．	LENGTH 〈変数名〉	
	SIZE	変数のサイズを，バイトを単位として返す．	SIZE 〈変数名〉	
	シフト・カウント(レコード・フィールド名)	そのフィールドの最下位ビットを，そのレコードの最下位まで右シフトで持っていくために必要なビット数を表す．	〈レコード・フィールド名〉	
	MASK	そのフィールドに対応するビットを 1，それ以外のビットを 0 にセットした，そのレコードと同じ幅を持つビットマスクを返す．	MASK 〈レコード・フィールド名〉	
	WIDTH	そのフィールドまたはレコードの幅をビット数で返す．	WIDTH |〈レコード・フィールド名〉| |〈レコード名〉|	

MS-DOS主要ファンクション一覧

〈注意〉 CP／Mコンパチなファンクション，およびネットワーク関係のファンクションを除く

番号	機能	コール	リターン
00H	プログラムの終了	AH ←00H CS ←PSPのパラグラフ番号	なし
01H	キーボード入力とコンソールへのエコー	AH ←01H	AL ←入力された文字コード
02H	文字の出力	AH ←02H DL ←標準出力に出力する文字コード	なし
03H	AUX入力	AH ←03H	AL ←AUXから入力された文字コード
04H	AUX出力	AH ←04H DL ←AUXに出力する文字コード	なし
05H	プリンタ出力	AH ←05H DL ←プリンタに出力する文字コード	なし
06H	コンソールとの直接入出力	入力する場合 AH ←06H DL ←FFH	ゼロフラグがセットされた場合 AL ←00H(入力なし) ゼロフラグがリセットされた場合 AL ←入力された文字コード
		出力する場合 AH ←06H DL ←出力する文字(FFH以外)	なし
07H	コンソールからの直接入力	AH ←07H	AL ←標準入力から入力された文字コード
08H	エコーなしのキーボード入力	AH ←08H	AL ←標準入力から入力された文字コード
09H	文字列の出力	AH ←09H DS：DX ←標準出力に出力する文字列の先頭アドレス(文字列の終わりは"$"で識別する)	なし
0AH	バッファード・キーボード入力	AH ←0AH DS：DX ←入力バッファの先頭アドレス	なし
0BH	キーボードのステータスチェック	AH ←0BH	AL＝00H タイプアヘッドバッファは空である AL＝FFH タイプアヘッドバッファに文字がはいっている
0CH	バッファを空にしてキーボード入力	AH ←0CH AL ←01H, 06H, 07H, 08H, 0AH ：タイプアヘッドバッファを空にしたあと，対応する値のファンクションコールが行われる AL ←他の値 ：タイプアヘッドバッファを空にする	ALレジスタにファンクション番号を指定した場合，対応するファンクションコールに従う ファンクション番号を指定しない場合 AL ←00H
0DH	ディスクのリセット	AH ←0DH	なし
0EH	ドライブの選択	AH ←0EH DL ←ドライブ番号(0＝A, 1＝B, …)	AL ←MS-DOSが使用可能な最大のドライブ数(CONFIG.SYSのLASTDRIVEコマンドで設定した値)
19H	カレント・ドライブ番号の読出し	AH ←19H	AL＝現在選択されているドライブ(0＝A, 1＝B, …)
1BH	カレント・ドライブデータの取得	AH ←1BH	AL ←1クラスタあたりのセクタ数(アロケーションユニット) CX ←1セクタあたりのバイト数 DX ←1ドライブあたりのクラスタ数 DS：DX ←FAT IDのアドレス

番号	機能	コール	リターン
1CH	ドライブデータの取得	AH ←1CH DL ←ドライブ番号(0：カレント・ドライブ，1：A,…)	AL＝FFH　ドライブ番号の指定が無効である AL＝FFH 以外 AL ←1クラスタあたりのセクタ数(アロケーションユニット) CX ←1セクタあたりのバイト数 DX ←1ドライブあたりのクラスタ数 DS：BX ←FAT ID のアドレス
25H	割込みベクトルの設定	AH ←25H AL ←割込みベクトル番号 DS：DX ←割込み処理ルーチンのエントリ・アドレス	なし
26H	PSP の作成	AH ←26H DX ←作成する PSP のパラグラフ番号	なし
29H	ファイル名の解析	AH ←29H AL ←解析の制御(ファイル名解析の制御ビット参照) DS：SI ←解析する文字列の先頭アドレス ES：DI ←オープンされていない FCB の先頭アドレス	AL＝00H　ワイルドカードが使用されていない AL＝01H　ワイルドカードが使用されている AL＝FFH　ドライブ名が無効である DS：SI ←解析された文字列のなかのファイル名の直後のアドレス
2AH	日付の読出し	AH ←2AH	CX ←年(1980～2079) DH ←月(1～12) DL ←日(1～31) AL ←曜日(0：日，1：月，…6：土)
2BH	日付の設定	AH ←2BH CX ←年(1980～2079) DH ←月(1～12) DL ←日(1～31)	AL＝00H　日付がセットされた AL＝FFH　無効な日付である
2CH	時刻の読出し	AH ←2CH	CH ←時(0～23) CL ←分(0～59) DH ←秒(0～59) DL ←1／100 秒(0～99)
2DH	時刻の設定	AH ←2DH CH ←時(0～23) CL ←分(0～59) DH ←秒(0～59) DL ←1／100 秒(0～99)	AL＝00H　時刻がセットされた AL＝FFH　無効な時刻である
2EH	ベリファイフラグの設定	AH ←2EH AL：ビット0 ・0ベリファイを行わない ・1ベリファイを行う	なし
30H	DOS のバージョン番号の読出し	AH ←30H	AH ←バージョン番号の小数部 AL ←バージョン番号の整数部 BH ←OEM 番号 BL：CX(24 ビット)←ユーザー番号 }これらの返す値に関しては，現在未定義である
31H	プログラムの常駐終了	AH ←31H AL ←リターンコード(子プロセスから親プロセスへ渡す1バイトのデータ) DX ←メモリに常駐されるプログラムのパラグラフサイズ	なし
33H	ブレークチェックの制御	ブレークチェックの設定を得る場合(コード 00H) AH ←33H AL ←00H	AL＝FFH 以外 ブレークチェックの設定が得られた DL ←現在のブレークチェックの設定(0＝OFF，1＝ON) AL＝FFH　AL レジスタの値が 00H～01H の範囲外である
		ブレークチェックを設定する場合(コード 01H) AH ←33H AL ←01H DL：ビット0←0 ブレークチェック OFF ←1 ブレークチェック ON	AL＝FFH 以外 ブレークチェックの設定が行われた AL＝FFH　AL レジスタの値が 00H～01H の範囲外である
35H	割込みベクトルの読出し	AH ←35H AL ←割込みベクトルの番号	ES：BX ←割込みベクトル

番号	機能	コール	リターン
36H	ディスクの残り容量の読出し	AH ←36H DL ←ドライブ番号(0:カレント・ドライブ, 1:A, 2:B, …)	BX ←使用可能なクラスタ数 DX ←1ドライブあたりのクラスタ数 CX ←1セクタあたりのバイト数 AX ←1クラスタあたりのセクタ数 AX=FFFFH ドライブ番号が無効である
38H	国別情報の読出し	情報を読出す場合 AH ←38H AL ←情報を得ようとする国のカントリーコード(FFH以外) (00H:現在設定されている国, 01H:USA, 51H:日本) AL ←FFH(2バイトのカントリーコードを設定する場合)(Ver3.1で追加) BX ←カントリーコード(Ver3.1で追加) DS:DX ←返される情報に対する32バイトのバッファの先頭アドレス	キャリーがセットされなかった場合 指定したバッファに, 国際適用業務に該当する情報がセットされた キャリーがセットされた場合 AX=02H 指定された国のテーブルが存在しない
		現在の国を変更する場合 AH ←38H AL ←設定しようとする国のカントリーコード(FFH以外) AL ←FFH(2バイトのカントリーコードを設定する場合)(Ver3.1で追加) BX ←カントリーコード(Ver3.1で追加) DX ←FFFFH	キャリーがセットされなかった場合 ALレジスタで指定した国が, 現在の国に設定された キャリーがセットされた場合 AX=02H 無効なカントリーコード
39H	サブ・ディレクトリの作成	AH ←39H DS:DX ←パス名の先頭アドレス	キャリーがセットされなかった場合 ディレクトリが作成された キャリーがセットされた場合 AX=03H 指定されたパス名が無効であるか, またはパスが存在しない AX=05H ルート・ディレクトリに空き領域がないか, すでに同名のファイルまたはディレクトリが存在しているので, ディレクトリを作成することができない
3AH	サブ・ディレクトリの削除	AH ←3AH DS:DX ←パス名の先頭アドレス	キャリーがセットされなかった場合 ディレクトリが削除された キャリーがセットされた場合 AX=03H 指定されたパス名が無効であるか, またはパスが存在しない AX=05H 指定されたディレクトリが空でなかったか, ディレクトリではないか, またはルート・ディレクトリであったために削除ができなかった AX=10H カレント・ディレクトリを削除しようとした
3BH	カレント・ディレクトリの変更	AH ←3BH DS:DX ←パス名の先頭アドレス	キャリーがセットされなかった場合 カレント・ディレクトリが変更された キャリーがセットされた場合 AX=03H 指定されたパス名が無効であるか, またはパスが存在しない
3CH	ファイルの作成	AH ←3CH DS:DX ←パス名の先頭アドレス CX ←ファイル属性	キャリーがセットされなかった場合 AX ←ファイル・ハンドル キャリーがセットされた場合 AX=03H 指定されたパス名が無効であるか, またはパスが存在しない AX=04H オープンされているファイルが多すぎるファイルは作成されたが, アクセスのためのファイル・ハンドルに余裕がない, または内部システムテーブルに空き領域がない AX=05H 属性の指定が作成不可能なもの(ディレクトリ, ボリュームラベル)であった, またはファイルを保護する属性が与えられた

番号	機能	コール	リターン
3DH	ファイル／デバイスのオープン	AH ←3DH AL ←アクセス制御コード DS：DX ←パス名の先頭アドレス ＊アクセスコード参照	キャリーがセットされなかった場合 AX ←ファイル・ハンドル キャリーがセットされた場合 AX＝01H 無効な機能コード AX＝02H ファイルが存在しない AX＝03H 指定されたパス名が無効であるか，またはパスが存在しない AX＝04H オープンされているファイルが多すぎるファイルは作成されたが，アクセスのためのファイル・ハンドルに余裕がない，または内部システムテーブルに空き領域がない AX＝05H ディレクトリ，ボリュームラベルをオープンしようとした，あるいは読出し専用のファイルを書込み用にオープンしようとした AX＝0CH アクセスコードが00H〜02Hの範囲外である
3EH	ファイルのクローズ	AH ←3EH BX ←ファイル・ハンドル	キャリーがセットされなかった場合 ファイルはクローズされた キャリーがセットされた場合 AX＝06H 指定されたファイル・ハンドルは現在オープンされていない
3FH	ファイル／デバイスの読出し	AH ←3FH DS：DX ←入力バッファの先頭アドレス CX ←読み込むバイト数 BX ←ファイル・ハンドル	キャリーがセットされなかった場合 AX ←読み込まれたバイト数 キャリーがセットされた場合 AX＝05H 指定されたファイル・ハンドルは，読出しが許可されていない AX＝06H 指定されたファイル・ハンドルは，現在オープンされていない
40H	ファイル／デバイスの書込み	AH ←40H DS：DX ←出力バッファの先頭アドレス CX ←書き込むバイト数 BX ←ファイル・ハンドル	キャリーがセットされなかった場合 AX ←書き込まれたバイト数 キャリーがセットされた場合 AX＝05H 指定されたファイル・ハンドルは，書込みが許可されていない AX＝06H 指定されたファイル・ハンドルは，現在オープンされていない
41H	ファイルの削除	AH ←41H DS：DX ←パス名の先頭アドレス	キャリーがセットされなかった場合 ファイルは削除された キャリーがセットされた場合 AX＝02H ファイルが存在しない AX＝03H 指定されたパス名が無効であるか，またはパスが存在しない AX＝05H 指定されたパスがディレクトリ，または読出し専用のファイルである
42H	ファイル・ポインタの移動	AH ←42H CX：DX ←オフセット(CX×10000H＋DX)バイト AL ←00H ファイルの先頭からオフセットの位置に移動する 01H 現在の位置からオフセットを加えた位置に移動する 02H ファイルの終わりにオフセットを加えた位置に移動する BX ←ファイル・ハンドル	キャリーがセットされなかった場合 DX：AX ←移動されたファイル・ポインタのファイルの先頭からのオフセット キャリーがセットされた場合 AX＝01H ALレジスタの値が00H〜02Hの範囲外である AX＝06H 指定されたファイル・ハンドルは，現在オープンされていない
43H	ファイル属性の変更	ファイル属性を得る場合(コード00H) AH ←43H DS：DX ←パス名の先頭アドレス AL ←00H ファイル属性をセットする場合(コード01H) AH ←43H DS：DX ←パス名の先頭アドレス AL ←01H CX ←ファイル属性＊ ＊ファイル属性参照	キャリーがセットされなかった場合 エラーなし(ファイル属性をセットする場合) CX ←ファイル属性(ファイル属性を得る場合) キャリーがセットされた場合 AX＝01H ALレジスタの値が00H〜01Hの範囲外である AX＝03H 指定されたパス名が無効であるか，またはパスが存在しない AX＝05H 指定されたファイル属性に，変更できないものが含まれていた(ディレクトリ，ボリュームラベル)

番号	機　　能	コール	リターン
44H	デバイスに対する I/O コントロール	IOCTL データを得る(コード 00H) AH ←44H AL ←00H BX ←ファイル・ハンドル	キャリーがセットされなかった場合 DX ←デバイス情報 キャリーがセットされた場合 AX＝01H　無効な機能コード AX＝06H　指定されたファイル・ハンドルは，現在オープンされていない
		IOCTL データをセットする(コード 01H) AH ←44H AL ←01H BX ←ファイル・ハンドル DX ←デバイスデータ(DH＝0)	キャリーがセットされなかった場合 デバイス情報がセットされた キャリーがセットされた場合 AX＝01H　無効な機能コード AX＝06H　指定されたファイル・ハンドルは，現在オープンされていない
		IOCTL 文字列を受け取る(コード 02H) AH ←44H AL ←02H BX ←ファイル・ハンドル CX ←コントロール文字列のバイト数 DS : DX ←バッファの先頭アドレス	キャリーがセットされなかった場合 AX ←読出しまたは書込みの行われたバイト数 キャリーがセットされた場合 AX＝01H　無効な機能コード AX＝06H　指定されたファイル・ハンドルは現在オープンされていない
		IOCTL 文字列を送る(コード 03H) AH ←44H AL ←03H BX ←ファイル・ハンドル CX ←コントロール文字列のバイト数 DS : DX ←バッファの先頭アドレス	
		IOCTL 文字列をドライブから受け取る(コード 04H) AH ←44H AL ←04H BL ←ドライブ番号(0 : カレント・ドライブ，1 : A, 2 : B, …) CX ←コントロール文字列のバイト数 DS : DX ←バッファの先頭アドレス	キャリーがセットされなかった場合 AX ←読出しまたは書込みの行われたバイト数 キャリーがセットされた場合 AX＝01H　無効な機能コード AX＝05H　アクセスが拒否された
		IOCTL 文字列をドライブに送る(コード 05H) AH ←44H AL ←05H BL ←ドライブ番号(0 : カレント・ドライブ，1 : A, 2 : B, …) CX ←コントロール文字列のバイト数 DS : DX ←バッファの先頭アドレス	
		入力ステータスを得る(コード 06H) AH ←44H AL ←06H BX ←ファイル・ハンドル	キャリーがセットされなかった場合 AL＝00H　読出しまたは書込みが行えない AL＝FFH　読出しまたは書込みが行える キャリーがセットされた場合 AX＝01H　無効な機能コード AX＝05H　アクセスが拒否された AX＝06H　指定されたファイル・ハンドルは現在オープンされていない AX＝0DH　指定されたデバイス情報が無効である
		出力ステータスを得る(コード 07H) AH ←44H AL ←07H BX ←ファイル・ハンドル	
		メディアの交換性(コード 08H) AH ←44H AL ←08H BL ←ドライブ番号(0 : カレント・ドライブ，1 : A, 2 : B, …) (Ver3.1で追加)	キャリーがセットされなかった場合 AX ← 00H　メディアが交換可能である AX ← 01H　メディアの交換が不可能である キャリーがセットされた場合 AX＝01H　デバイス・ドライバがこの機能をサポートしていない AX＝0FH　指定されたドライブ番号が無効である
45H	ファイル・ハンドルのコピー	AH ←45H BX ←ファイル・ハンドル	キャリーがセットされなかった場合 AX ←コピーされたファイル・ハンドル キャリーがセットされた場合 AX＝04H　オープンされているファイルが多すぎるまたは内部システムテーブルに空き領域がない AX＝06H　指定されたファイル・ハンドルは現在オープンされていない

番号	機能	コール	リターン
46H	指定したファイル・ハンドルへのコピー	AH ←46H BX ←既存のファイル・ハンドル CX ←新規に作成するファイル・ハンドル	キャリーがセットされなかった場合 エラーなし キャリーがセットされた場合 AX＝04H　オープンされているファイルが多すぎるまたは内部システムテーブルに空き領域がない AX＝06H　指定されたファイル・ハンドルは現在オープンされていない
47H	カレント・ディレクトリの読出し	AH ←47H DS：SI ←返される情報に対する64バイトのバッファの先頭アドレス DL ←ドライブ番号(0：カレント・ドライブ，1：A，2：B，…)	キャリーがセットされなかった場合 エラーなし キャリーがセットされた場合 AX＝0FH　ドライブ番号が無効である(バッファの内容は変更されない)
48H	メモリブロックの割当て	AH ←48H BX ←割当てを要求するメモリ領域のパラグラフサイズ	キャリーがセットされなかった場合 AX ←割り当てられたメモリの先頭のパラグラフ番号 キャリーがセットされた場合 AX＝07H　メモリ・コントロールブロックが破壊されている AX＝08H　十分な大きさのメモリがない BX ←割当て可能な最大のメモリ領域のパラグラフサイズ
49H	メモリブロックの開放	AH ←49H ES ←開放するメモリ領域のパラグラフ番号	キャリーがセットされなかった場合 エラーなし キャリーがセットされた場合 AX＝07H　メモリ・コントロールブロックが破壊されている AX＝09H　指定されたメモリ領域は，メモリアロケーションによって割り当てられたものではない
4AH	メモリブロックのサイズの変更	AH ←4AH ES ←メモリ領域のパラグラフ番号 BX ←新たなメモリ領域のパラグラフサイズ	キャリーがセットされなかった場合 エラーなし キャリーがセットされた場合 AX＝07H　メモリ・コントロールブロックが破壊されている AX＝08H　十分な大きさのメモリがない BX ←割当て可能な最大のメモリ領域のパラグラフサイズ AX＝09H　指定されたメモリ領域は，メモリアロケーションによって割り当てられたものではない
4BH	プログラムのロードと実行	AH ←4BH DS：DX ←対象とするファイルのパス名の先頭アドレス ES：BX ←パラメータブロック*の先頭アドレス AL ←00H プログラムを読み込み，実行する ←03H プログラムを読み込む ＊パラメータブロック参照	キャリーがセットされなかった場合 エラーなし キャリーがセットされた場合 AX＝02H　ファイルが存在しない AX＝03H　指定されたパス名が無効であるか，またはパスが存在しない AX＝08H　十分な大きさのメモリがない AX＝0AH　環境変数が32Kバイト以上ある AX＝0BH　指定されたファイルのヘッダが正しくない(EXEファイルのみ)
4CH	プロセスの終了	AH ←4CH AL ←リターンコード(子プロセスから親プロセスへ渡す1バイトのデータ)	なし
4DH	子プロセスの終了コードの読出し	AH ←4DH	AH＝00H　INT 20H，ファンクション00H，4CHによる終了 AL ←子プロセスのリターンコード AH＝01H　CTRL-Cによる終了(INT 23H) AH＝02H　致命的エラー(INT 24H) AH＝03H　常駐したまま終了(INT 27H，ファンクション31H) AL ←子プロセスからのリターンコード
4EH	一致するファイルの検索	AH ←4EH DS：DX ←パス名の先頭アドレス CX ←ファイル属性	キャリーがセットされなかった場合 DTAに検索されたファイルの情報が書き込まれる キャリーがセットされた場合 AX＝02H　ファイルが存在しない AX＝12H　これ以上ファイルが存在しない
4FH	次に一致するファイルの検索	AH ←4FH DS：DX ←パス名の先頭アドレス CX ←ファイル属性	キャリーがセットされなかった場合 DTAに検索されたファイルの情報が書き込まれる キャリーがセットされた場合 AX＝12H　これ以上ファイルが存在しない

番号	機能	コール	リターン
54H	ベリファイフラグの読出し	AH ←54H	AL ←現在のベリファイフラグの値(0＝OFF, 1＝ON)
56H	ディレクトリ・エントリの移動	AH ←56H DS：DX ←パス名の先頭アドレス ES：DI ←移動先のパス名の先頭アドレス	キャリーがセットされなかった場合 エラーなし キャリーがセットされた場合 AX＝02H　DS：DX で指定されたファイルが存在しない AX＝03H　指定されたパス名が無効であるか，またはパスが存在しない AX＝05H　DS：DX で指定されたパス名がディレクトリであったかES：DI で指定されたファイルがすでに存在している AX＝11H　違うドライブ間で移動しようとした
57H	ファイルの時間情報の読出し／設定	日付／時刻を得る(コード 00H) AH ←57H AL ←00H BX ←ファイル・ハンドル	キャリーがセットされなかった場合 DX／CX ←ファイルが最後に編集された日付／時刻 キャリーがセットされた場合 AX＝01H　AL レジスタの値が 00H～01H の範囲外である AX＝06H　指定されたファイル・ハンドルは，現在オープンされていない
		日付／時刻をセットする(コード 01H) AH ←57H AL ←01H BX ←ファイル・ハンドル CX ←セットする時刻 DX ←セットする日付	キャリーがセットされなかった場合 エラーなし(ファイルがクローズされるときまで記録されない) キャリーがセットされた場合 AX＝01H　AL レジスタの値が 00H～01H の範囲外である AX＝06H　指定されたファイル・ハンドルは，現在オープンされていない
58H	メモリ割当て方法の設定／取得	ストラテジを得る(コード 00H) AH ←58H AL ←00H	キャリーがセットされなかった場合 AX ←ストラテジ(00H：下位, 01H：最小, 02H：上位) キャリーがセットされた場合 AX＝01H　AL レジスタの値が無効であるか，または設定したストラテジが 00H～02H の範囲外である
		ストラテジを設定する(コード 01H) AH ←58H AL ←01H BX ←ストラテジ(00H：下位，01H：最小，02H：上位) ＊ストラテジ参照	キャリーがセットされなかった場合 エラーなし キャリーがセットされた場合 AX＝01H　AL レジスタの値が無効であるか，または設定したストラテジが 00H～02H の範囲外である
59H	拡張エラーコードの取得	AH ←59H	AX ←拡張エラーコード BH ←エラークラス BL ←可能な対処 CH ←エラーの発生箇所 CL, DX, SI, DI, BP, DS, ES の各レジスタは破壊される
5AH	テンポラリファイルの作成 (Ver3.1で追加)	AH ←5AH CX ←ファイル属性 DS：DX ←パス名の先頭アドレス(パス名の後ろに 13 バイトのファイル名を格納する領域を確保する必要がある)	キャリーがセットされなかった場合 AX ←ファイル・ハンドル キャリーがセットされた場合 AX＝03H　DS：DX で指定したディレクトリが無効か，または存在しない AX＝05H　アクセスが否定された
5BH	新規ファイルの作成 (Ver3.1で追加)	AH ←5BH CX ←ファイル属性 DS：DX ←パス名の先頭アドレス ＊ファイル属性参照	キャリーがセットされなかった場合 AX ←ファイル・ハンドル キャリーがセットされた場合 AX＝03H　DS：DX で指定したディレクトリが無効か，または存在しない AX＝04H　カレント・プロセス中に利用可能なハンドルがないか，または MS-DOS のシステムテーブルに余裕がない AX＝05H　属性の指定が作成不可能なもの(ディレクトリ, ボリュームラベル)がはいっていたか，または同じ名前のディレクトリが存在した AX＝50H　DS：DX で指定したファイルがすでに存在する
62H	PSP の取得 (Ver3.1で追加)	AH ←62H	BX ←カレント・プロセスの PSP のセグメント・アドレス

ファイル名解析の制御ビット

ビット	値	機能
0	0	ファイル区切り記号を検出した場合，すべての解析を停止させる．
	1	先行するファイル区切り記号は無視する．
1	0	文字列にドライブ番号がはいっていない場合，FCBのドライブ番号(FCBのオフセット00H）には00H(カレントドライブ)がセットされる．
	1	文字列にドライブ番号がはいっていない場合，FCBのドライブ番号は変更されない．
2	0	文字列にファイル名がはいっていない場合，FCBのファイル名には8つのスペースがセットされる．
	1	文字列にファイル名がはいっていない場合，FCBのファイル名は変更されない．
3	0	文字列に拡張子がはいっていない場合，FCBの拡張子には3つのスペースがセットされる．
	1	文字列に拡張子がはいっていない場合，FCB内の拡張子は変更されない．

検索されたファイルの情報

DTA＋オフセット	検索されたファイルの情報
DTA＋00H	検索するファイルの属性
DTA＋01H	ドライブ番号（0：A，1：B，……）
DTA＋02H～＋09H	検索するパス名のなかのファイル名
DTA＋0AH～＋0CH	検索するパス名のなかの拡張子
DTA＋0DH～＋14H	システム予約
DTA＋15H	検索されたファイルの属性
DTA＋16H～＋17H	time
DTA＋18H～＋19H	date
DTA＋1AH～＋1DH	ファイルの大きさ（4バイト）
DTA＋1EH～＋2AH	パックネーム(packed name)（ASCIZ文字列） （ファイル名が8文字未満の場合に，スペースが除かれたもの）
例：パックネーム	ABC（名前） .XYZ（拡張子） →ABC.XYZ00H（packed name）

ファイル属性

属性	意味
01H	読出し専用ファイル
02H	不可視属性，通常のディレクトリサーチから除外される
04H	システムファイル
08H	ボリュームラベル
10H	ディレクトリ
20H	保存ビット

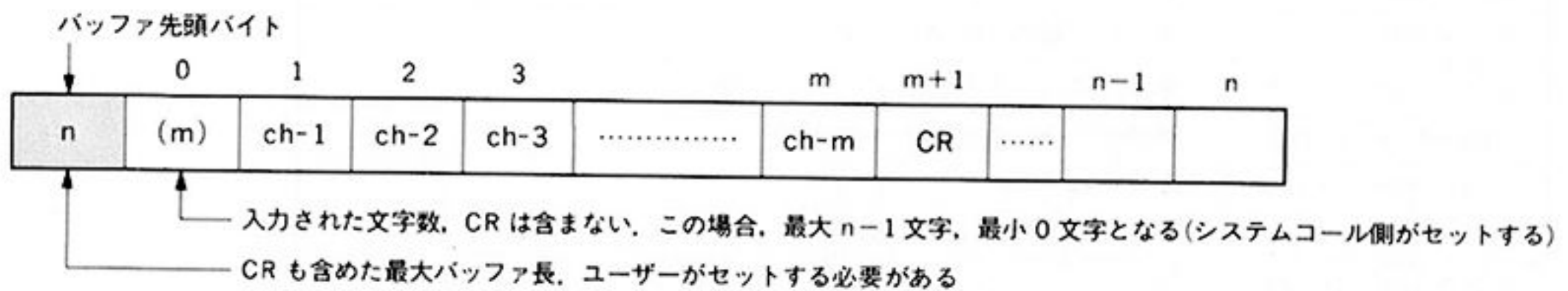
パラメータブロック

●AL＝00Hの場合

オフセット	サイズ	内容
＋00H	1ワード	環境のセグメントアドレス
	2ワード	PSPのオフセット80Hに作成されるコマンドラインに与える80Hバイトのデータのポインタ
	2ワード	PSPのオフセット5CHに作成されるデフォルトFCBに与える10Hバイトのデータのポインタ
	2ワード	PSPのオフセット6CHに作成されるデフォルトFCBに与える10Hバイトのデータのポインタ

●AL＝03Hの場合

オフセット	サイズ	内容
＋00H	1ワード	ファイルをロードするセグメントアドレス
＋02H	1ワード	イメージに対して使用される再配置因子

入力用バッファ

アクセスコード

ビット0～3	アクセスコード
0000	読出し
0001	書込み
0010	読出し／書込み

ストラテジの意味

ストラテジ	意　　味
00H(下位)	最も下位に位置する利用可能なメモリブロックを割り当てる(デフォルト)
01H(最小)	必要最小のメモリブロックを割り当てる
02H(上位)	最も上位に位置する利用可能なメモリブロックを割り当てる

索引

A
ASSUME 擬似命令 ………… 88, 132

B
BP レジスタ ………… 147, 248
BYTE ………… 224
BYTE PTR ………… 84

C
CLI 命令 ………… 57
CMPS 命令 ………… 150
COM モデル ………… 121, 172
CS レジスタ ………… 119
C 言語 ………… 52, 243, 270

D
DB 擬似命令 ………… 73
DD 擬似命令 ………… 77
DQ 擬似命令 ………… 77
DS レジスタ ………… 119
DT 擬似命令 ………… 77
DUP 擬似命令 ………… 75
DW 擬似命令 ………… 76

E
ELSE 擬似命令 ………… 190
ENDIF 擬似命令 ………… 190
ENDM 擬似命令 ………… 204
ENDP 擬似命令 ………… 181
ENDS 擬似命令 ………… 87, 127
END 擬似命令 ………… 71
EOI の発行 ………… 276
EQU 擬似命令 ………… 177, 200
ES レジスタ ………… 142
EXE2BIN コマンド ………… 98, 173
EXE ファイルの構造 ………… 170
EXE ヘッダ ………… 170
EXE モデル ………… 121, 160, 172
EXTRN 擬似命令 ………… 223

F
FAR ………… 187, 224

G
GROUP 擬似命令 ………… 139, 152

I
IFDIF 擬似命令 ………… 209
IF 擬似命令 ………… 190
INCLUDE 擬似命令 ………… 180
IRET 命令 ………… 57, 275
I/O 空間 ………… 28
I/O ポート ………… 27

L
LDS 命令 ………… 252
LES 命令 ………… 252
LIB コマンド ………… 220, 239, 315
LINK コマンド ………… 97, 220, 237, 312
LOCAL 擬似命令 ………… 210
LODS 命令 ………… 150

M

MACRO 擬似命令…………………… 204
MASM コマンド……………… 93, 310
MOVS 命令 …………………… 150
MS-DOS 互換モード …………… 306
MS-DOS の役割 ……………… 37, 39
MS-Windows …………………… 50

N

NEAR……………………… 187, 224
NOP 命令 ……………………… 103

O

OFFSET 演算子 ……………… 82, 152
ORG 擬似命令 ………………… 69
OS ……………………………… 15, 40
OS/2 ………………………… 50, 302

P

PHASE ERROR ………………… 146
PRIVATE 属性……………………… 226
PROC 擬似命令……………………… 181
PSP ………………………………… 173
PTR 演算子 ………………… 83, 190
PUBLIC 擬似命令…………………… 222
PUBLIC 属性……………………… 226

R

ROM BIOS ………………………… 35

S

SCAS 命令 ………………………… 150
SEGMENT 擬似命令………… 87, 127
SP レジスタ ……………………… 148
SS レジスタ ……………………… 119
STACK 属性……………………… 226
STI 命令 ………………………… 57
STOS 命令 ……………………… 150
SYMDEB コマンド ……………… 98

V

VRAM ……………………………… 29
VSYNC 割り込み………………… 277

W

WORD ……………………………… 224
WORR PTR ……………………… 84

その他

1ラインアセンブラ ……………… 196
2パスアセンブラ ………………… 197
80286CPU ……………………… 302
8086CPU ………………………… 31
8086CPU のメモリ空間 ………… 110
α-β カット……………………… 257

ア

アセンブルエラー ……………… 94
アセンブルの操作 ……………… 21
アドレッシングモード ………… 147
エスケープコード ……………… 48
エスケープシーケンス …… 46, 48, 107
オブジェクトファイル ………… 93
オフセットアドレス …………… 115

カ

外部参照の解決 ………………… 236
仮想記憶 ………………………… 308
型属性 …………………… 187, 224
仮引数 …………………………… 207
環境セグメント ………………… 173
関数 ……………………………… 244
関数値 …………………………… 249
擬似命令 ………………………… 63
擬似命令一覧 …………………… 318
擬似命令の役割 ………………… 64

キーボード割り込み …………………… 277
キャラクタ型デバイス ………………… 292
高級言語 ………………………………… 15, 52
構造化 …………………………………… 181
コードセグメント ……………………… 116
コードラベル ……………………………… 79
コマンドパケット ……………………… 292
コメント …………………………………… 90
コメントフィールド ……………………… 90
コンバインタイプ ……………………… 226

サ

再配置情報 ……………………………… 236
サブルーチン …………………………… 181
システムコール …………………………… 40
実引数 …………………………………… 207
条件マクロ ……………………………… 209
常駐終了 ………………………………… 279
数値表現 …………………………………… 66
スタックセグメント ………… 116, 130
スタックポインタ ……………………… 148
スタック領域 …………………………… 148
ストラテジルーチン …………………… 291
ストリング命令 ………………………… 150
スモールモデル ………… 110, 189, 249
スワップアウト ………………………… 308
セクタ ……………………………………… 39
セグメントアドレス …… 114, 125, 302
セグメントオーバーライドプリフィックス …………………… 142, 144, 166
セグメントグループ …………………… 138
セグメントの結合 ……………………… 226
セグメントのリロケート ……………… 168
セグメント方式 ………………………… 112
セグメントレジスタ …………………… 118
前方参照 ………………… 85, 146, 190
相対アドレス ………………………… 67, 102

タ

タイマー割り込み ……………………… 277
ターゲットプログラム …………………… 99
タスク …………………………………… 306
ダミー引数 ……………………………… 207
定数定義 ………………………………… 177
ディスクリプタテーブルレジスタ 302
ディスプレイ画面の制御 ………… 29
データ型 …………………………………… 86
データセグメント ……………………… 116
データセグメントの選択 ………… 133
データラベル ……………………………… 78
デバイスドライバ ……………… 57, 290
トラック …………………………………… 39
トレース機能 …………………………… 162

ナ

ニアジャンプ …………………………… 120
ニーモニック ……………………………… 14

ハ

バイト型 …………………………………… 77
配列の参照 ………………………………… 80
バッファリング ………………………… 109
ハードウェア情報 ………………………… 34
ハードウェア割り込み ……… 56, 272
パブリック ……………………………… 222
パラグラフ ……………………………… 124
ハンドアセンブル ……………………… 195
引数 ……………………………………… 245
ヒープ …………………………………… 250
標準出力 ………………………………… 157
標準入力 ………………………………… 157
ファイル …………………………………… 40
ファイル形式の変換 ……………………… 23
ファイルハンドル ……………………… 157
ファーコール …………………………… 120
ファージャンプ ………………………… 120

ファームウェア ……………………… 36
ファーリターン ……………………… 120
ファンクションコール ……… 41, 178
ファンクションコール一覧 ……… 322
フィールド …………………………… 89
フェイズエラー ……………………… 146
フックをかける ……………………… 278
物理アドレス …………… 114, 123, 302
フレームポインタ …………………… 248
プログラム開発の手順 ……………… 17
プログラム実行の仕組み ………… 116
プログラムの部品化 ……………… 218
プロシージャ …………… 181, 211, 244
ブロック型デバイス ……………… 292
ブロック転送 ……………………… 208
プロテクトモード …………… 302, 304
分割アセンブル …………………… 216
分割アセンブルの仕組み ………… 235
ヘッダファイル ……………… 181, 256
変数 ………………………………… 79
ポインタ ………………… 81, 147, 252

マ

マクロアセンブラ ………………… 197
マクロ定義 ………………………… 204
マクロ展開 ………………………… 205
マクロパラメータ ………………… 207
マクロ命令 ………………… 199, 211
マクロ呼び出し …………………… 205
マルチタスク ……………………… 306
ミニマックス法 …………………… 257
命令フィールド …………………… 90
メインモジュール ………………… 216
メモリ ……………………………… 27
文字コードの定義 ………………… 75
モジュール ………………………… 220
文字列の定義 ……………………… 75

ヤ

ユーザー領域 ……………………… 110

ラ

ライブラリ ……………………… 54, 220
ライブラリアン …………………… 239
ラージモデル …………… 110, 189, 249
ラベル ……………………………… 66
ラベルフィールド ………………… 90
リアルモード ………………… 302, 304
リストファイル …………………… 95
領域の確保 ………………………… 76
リロケーション情報 ……………… 170
リロケータブルオブジェクト …… 236
リロケート ………………………… 167
リンカ …………………………… 18, 237
リンクの操作 ……………………… 22
レジスタ …………………………… 27
ローカル …………………………… 222
ローカル変数 ………………… 147, 247
論理アドレス ……………………… 303

ワ

ワークエリア ………………… 110, 148
ワード型 …………………………… 77
割り込みイネーブルフラグ ……… 58
割り込みコントローラ …………… 274
割り込み処理 ………………… 57, 275
割り込みベクタ ……………… 58, 273
割り込みベクタテーブル ………… 273
割り込みマスクレジスタ ………… 274
割り込みルーチン ………………… 291

参考文献

・「iAPX86 マクロアセンブリ言語プログラミングマニュアル」
インテルジャパン株式会社　CQ 出版

・「MS-DOS ユーザーズマニュアル」　日本電気株式会社

・「MS-DOS マクロアセンブラマニュアル」　日本電気株式会社

・「Microsoft Macro Assembler ユーザーズガイド&リファレンスマニュアル」
マイクロソフト株式会社

・「Microsoft C Compiler ユーザーズガイド」　マイクロソフト株式会社

・「Turbo C ユーザーズガイド」　株式会社マイクロソフトウェアアソシエイツ

・「C-MASM ユーザーズマニュアル」　株式会社ライフボート

・「実戦マクロアセンブラ活用法」　中野正次著　CQ 出版

・「FMR 徹底解析」　BNN 第 2 企画部編　株式会社ビー・エヌ・エヌ

・「PC-9800 シリーズ　テクニカルデータブック」
アスキー出版局テクライト編　アスキー出版局

・「MS-DOS アセンブラ MACRO プログラミング技法」
西村卓也著　アスキー出版局

・「MS-DOS デバイスドライバ活用技法」
株式会社エー・ピー・ラボ　日笠健著　アスキー出版局

・「標準 MS-DOS ハンドブック」　アスキー出版局編著　アスキー出版局

・「MS-DOS3.1 ハンドブック」　アスキー書籍編集部編著　アスキー出版局

・「MS-DOS プログラマーズハンドブック」
アスキー書籍編集部編著　アスキー出版局

・「80286 ハンドブック」　大貫広幸，田中恵介，蓑原隆共著　アスキー出版局

・「応用 MS-DOS」　村瀬康治著　アスキー出版局

・「応用 C 言語」　三田典玄著　アスキー出版局

・「思考ゲームプログラミング」
森田和郎，国枝交子，津田伸秀共著　アスキー出版局

・「日本語 OS/2 プレリリース・セミナー」
マイクロソフト株式会社監修　アスキー書籍編集部編　アスキー出版局

執筆者紹介

蒲地 輝尚（かまち てるひさ）

1964年長崎市に生まれ，鹿児島市で育つ．

東京大学工学部卒．現在，大手電気メーカー勤務．

執筆協力に「応用 MS-DOS」（アスキー出版局），著書に「はじめて読む 8086」（アスキー出版局）がある．

SMC-777，Apple II，PC-98LT，Macintosh II を所有．

ソフトウェアの開発環境やエンドユーザーのためのユーザーインターフェイスに興味を持っている．

プログラム協力

蔭山 哲也

中田 秀基

編集協力

藤本 衡

はじめて読むMASM

1988年9月1日　初版発行
1996年1月31日　第1版第22刷発行

著　者　蒲地 輝尚（かまち てるひさ）
発行人　宮崎 秀規
編集人　佐藤 英一
発行所　**株式会社アスキー**
〒151-24　東京都渋谷区代々木4-33-10
振　替　00140-7-161144
大代表　(03)5351-8111
出版営業部　(03)5351-8194（ダイヤルイン）
第一書籍編集部　(03)5351-8106（ダイヤルイン）

制　作　株式会社 GARO
印　刷　株式会社 加藤文明社印刷所

編　集　佐藤 英一

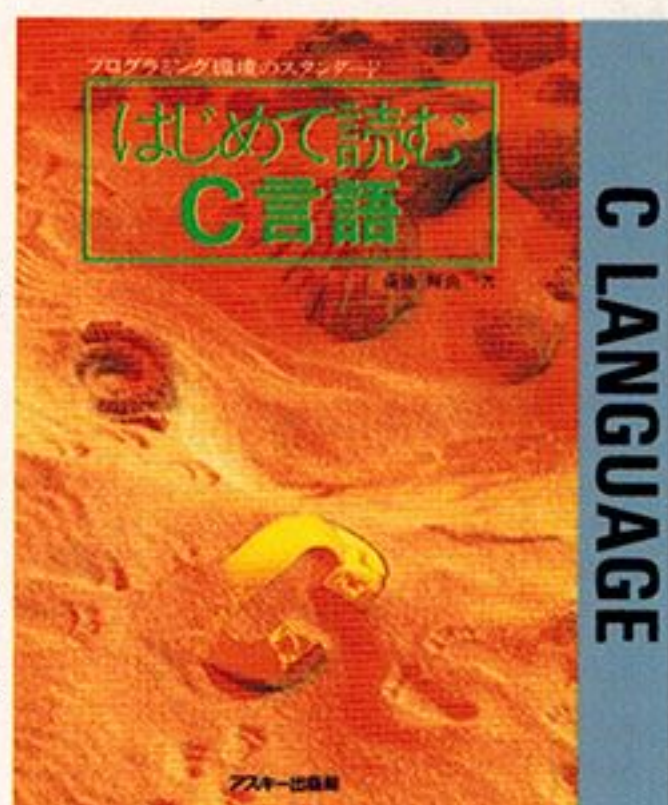
ISBN4-87148-313-4　　**Printed in Japan**

●11321／11421

はじめて読むシリーズ

はじめて読むC言語

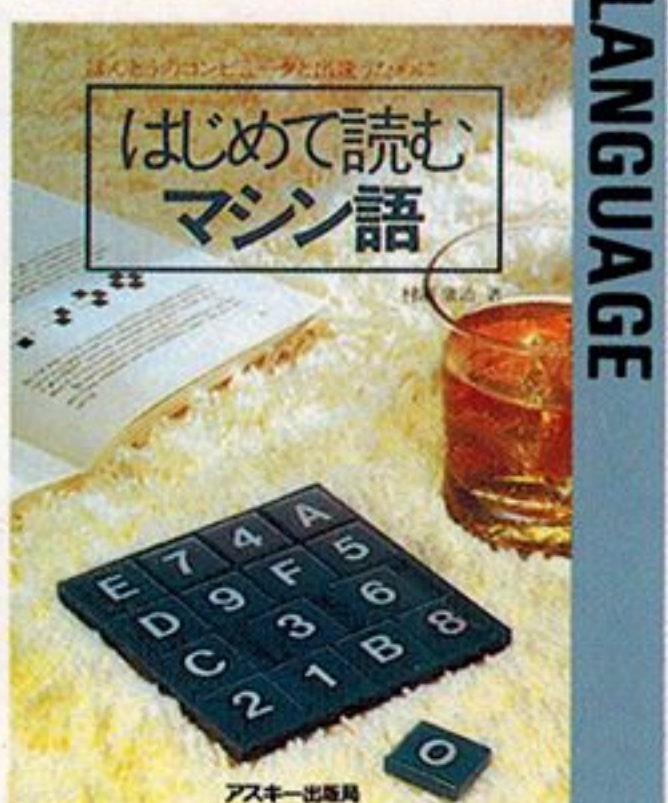
C LANGUAGE

蒲地輝尚 著　定価1,800円（本体1,748円）
標準的なプログラミング環境であるC言語を，その基礎から徹底解説．やさしい図説を読み進めるうちに，C言語は難しいと気後れしていたユーザーでも，自然にプログラムを組む力が身に付きます．全Cユーザー必読の1冊．

はじめて読む8086

MACHINE LANGUAGE

蒲地輝尚 著 村瀬康治 監修 定価1,650円（本体1,602円）
多くの16ビット・コンピュータで採用されている8086，V30，80286系CPUのマシン語を，MS-DOSの標準ツールを使ってやさしく実習．これからMS-DOSやアセンブラの上級にチャレンジしようとする読者には必須の書籍です．

はじめて読むマシン語

村瀬康治 著　　定価1,240円（本体1,204円）
はじめてマシン語を学ぶ人のための啓蒙的入門書．コンピュータの基礎知識もあわせて解説していますから，初心者でも十分に読みこなすことができます．PC-8801シリーズ，X1シリーズ，MSXなどZ80，8080系マシンユーザー必携．

はじめて読むアセンブラ

ASSEMBLER

村瀬康治 著　　定価1,650円（本体1,602円）
本書は「はじめて読むマシン語」の続編です．本格的なプログラム作りを目指す人のために，アセンブラの全体像をやさしく解説．CP/M上の各種ツールの使い方を学びながら，ソフトウェア開発の基礎をしっかりと把握できます．